LinksPlatform's Platform.Data.Doublets.Sequences Class Library

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

```csharp
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{TLinkAddress}"/>
    public class BalancedVariantConverter<TLinkAddress> :
        LinksListToSequenceConverterBase<TLinkAddress>
    {
        /// <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<TLinkAddress> 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 TLinkAddress Convert(IList<TLinkAddress> 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 TLinkAddress[(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]);
        }
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```csharp
        private void HalveSequence(IList<TLinkAddress> destination, IList<TLinkAddress> 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

```csharp
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<TLinkAddress> :
    ↪  LinksListToSequenceConverterBase<TLinkAddress>
    {
        private static readonly LinksConstants<TLinkAddress> _constants =
        ↪  Default<LinksConstants<TLinkAddress>>.Instance;
        private static readonly EqualityComparer<TLinkAddress> _equalityComparer =
        ↪  EqualityComparer<TLinkAddress>.Default;
        private static readonly Comparer<TLinkAddress> _comparer =
        ↪  Comparer<TLinkAddress>.Default;
        private static readonly TLinkAddress _zero = default;
        private static readonly TLinkAddress _one = Arithmetic.Increment(_zero);
        private readonly IConverter<IList<TLinkAddress>, TLinkAddress> _baseConverter;
        private readonly LinkFrequenciesCache<TLinkAddress> _doubletFrequenciesCache;
        private readonly TLinkAddress _minFrequencyToCompress;
        private readonly bool _doInitialFrequenciesIncrement;
        private Doublet<TLinkAddress> _maxDoublet;
        private LinkFrequency<TLinkAddress> _maxDoubletData;
        private struct HalfDoublet
        {
            /// <summary>
            /// <para>
            /// The element.
            /// </para>
            /// <para></para>
            /// </summary>
            public TLinkAddress Element;
            /// <summary>
            /// <para>
            /// The doublet data.
            /// </para>
            /// <para></para>
            /// </summary>
            public LinkFrequency<TLinkAddress> DoubletData;

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

```csharp
            /// <para></para>
            /// </param>
            /// <param name="doubletData">
            /// <para>A doublet data.</para>
            /// <para></para>
            /// </param>
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public HalfDoublet(TLinkAddress element, LinkFrequency<TLinkAddress> doubletData)
            {
                Element = element;
                DoubletData = doubletData;
            }

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

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

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

```csharp
        /// <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<TLinkAddress> links, IConverter<IList<TLinkAddress>,
            TLinkAddress> baseConverter, LinkFrequenciesCache<TLinkAddress>
            doubletFrequenciesCache, TLinkAddress 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 TLinkAddress Convert(IList<TLinkAddress> source) =>
            _baseConverter.Convert(Compress(source));
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private IList<TLinkAddress> Compress(IList<TLinkAddress> 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
```

```
204            var copy = new HalfDoublet[sequence.Count];
205            Doublet<TLinkAddress> doublet = default;
206            for (var i = 1; i < sequence.Count; i++)
207            {
208                doublet = new Doublet<TLinkAddress>(sequence[i - 1], sequence[i]);
209                LinkFrequency<TLinkAddress> data;
210                if (_doInitialFrequenciesIncrement)
211                {
212                    data = _doubletFrequenciesCache.IncrementFrequency(ref doublet);
213                }
214                else
215                {
216                    data = _doubletFrequenciesCache.GetFrequency(ref doublet);
217                    if (data == null)
218                    {
219                        throw new NotSupportedException("If you ask not to increment
                         ↪ frequencies, it is expected that all frequencies for the sequence
                         ↪ are prepared.");
220                    }
221                }
222                copy[i - 1].Element = sequence[i - 1];
223                copy[i - 1].DoubletData = data;
224                UpdateMaxDoublet(ref doublet, data);
225            }
226            copy[sequence.Count - 1].Element = sequence[sequence.Count - 1];
227            copy[sequence.Count - 1].DoubletData = new LinkFrequency<TLinkAddress>();
228            if (_comparer.Compare(_maxDoubletData.Frequency, default) > 0)
229            {
230                var newLength = ReplaceDoublets(copy);
231                sequence = new TLinkAddress[newLength];
232                for (int i = 0; i < newLength; i++)
233                {
234                    sequence[i] = copy[i].Element;
235                }
236            }
237            return sequence;
238        }
239        [MethodImpl(MethodImplOptions.AggressiveInlining)]
240        private int ReplaceDoublets(HalfDoublet[] copy)
241        {
242            var oldLength = copy.Length;
243            var newLength = copy.Length;
244            while (_comparer.Compare(_maxDoubletData.Frequency, default) > 0)
245            {
246                var maxDoubletSource = _maxDoublet.Source;
247                var maxDoubletTarget = _maxDoublet.Target;
248                if (_equalityComparer.Equals(_maxDoubletData.Link, _constants.Null))
249                {
250                    _maxDoubletData.Link = _links.GetOrCreate(maxDoubletSource,
                     ↪ maxDoubletTarget);
251                }
252                var maxDoubletReplacementLink = _maxDoubletData.Link;
253                oldLength--;
254                var oldLengthMinusTwo = oldLength - 1;
255                // Substitute all usages
256                int w = 0, r = 0; // (r == read, w == write)
257                for (; r < oldLength; r++)
258                {
259                    if (_equalityComparer.Equals(copy[r].Element, maxDoubletSource) &&
                     ↪ _equalityComparer.Equals(copy[r + 1].Element, maxDoubletTarget))
260                    {
261                        if (r > 0)
262                        {
263                            var previous = copy[w - 1].Element;
264                            copy[w - 1].DoubletData.DecrementFrequency();
265                            copy[w - 1].DoubletData =
                             ↪ _doubletFrequenciesCache.IncrementFrequency(previous,
                             ↪ maxDoubletReplacementLink);
266                        }
267                        if (r < oldLengthMinusTwo)
268                        {
269                            var next = copy[r + 2].Element;
270                            copy[r + 1].DoubletData.DecrementFrequency();
271                            copy[w].DoubletData = _doubletFrequenciesCache.IncrementFrequency(ma↵
                             ↪ xDoubletReplacementLink,
                             ↪ next);
272                        }
273                        copy[w++].Element = maxDoubletReplacementLink;
```

```csharp
                            r++;
                            newLength--;
                        }
                        else
                        {
                            copy[w++] = copy[r];
                        }
                    }
                    if (w < newLength)
                    {
                        copy[w] = copy[r];
                    }
                    oldLength = newLength;
                    ResetMaxDoublet();
                    UpdateMaxDoublet(copy, newLength);
                }
                return newLength;
            }
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            private void ResetMaxDoublet()
            {
                _maxDoublet = new Doublet<TLinkAddress>();
                _maxDoubletData = new LinkFrequency<TLinkAddress>();
            }
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            private void UpdateMaxDoublet(HalfDoublet[] copy, int length)
            {
                Doublet<TLinkAddress> doublet = default;
                for (var i = 1; i < length; i++)
                {
                    doublet = new Doublet<TLinkAddress>(copy[i - 1].Element, copy[i].Element);
                    UpdateMaxDoublet(ref doublet, copy[i - 1].DoubletData);
                }
            }
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            private void UpdateMaxDoublet(ref Doublet<TLinkAddress> doublet,
                LinkFrequency<TLinkAddress> data)
            {
                var frequency = data.Frequency;
                var maxFrequency = _maxDoubletData.Frequency;
                //if (frequency > _minFrequencyToCompress && (maxFrequency < frequency ||
                //    (maxFrequency == frequency && doublet.Source + doublet.Target < /* gives better
                //    compression string data (and gives collisions quickly) */ _maxDoublet.Source +
                //    _maxDoublet.Target)))
                if (_comparer.Compare(frequency, _minFrequencyToCompress) > 0 &&
                    (_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) */
                {
                    _maxDoublet = doublet;
                    _maxDoubletData = data;
                }
            }
        }
    }
}
```

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

```csharp
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 links list to sequence converter base.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="LinksOperatorBase{TLinkAddress}"/>
    /// <seealso cref="IConverter{IList{TLinkAddress}, TLinkAddress}"/>
    public abstract class LinksListToSequenceConverterBase<TLinkAddress> :
        LinksOperatorBase<TLinkAddress>, IConverter<IList<TLinkAddress>, TLinkAddress>
    {
```

```csharp
        /// <summary>
        /// <para>
        /// Initializes a new <see cref="LinksListToSequenceConverterBase"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="links">
        /// <para>A links.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected LinksListToSequenceConverterBase(ILinks<TLinkAddress> links) : base(links) { }

        /// <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 abstract TLinkAddress Convert(IList<TLinkAddress> source);
    }
}
```

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

```csharp
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{TLinkAddress}"/>
    public class OptimalVariantConverter<TLinkAddress> :
    ↪   LinksListToSequenceConverterBase<TLinkAddress>
    {
        private static readonly EqualityComparer<TLinkAddress> _equalityComparer =
        ↪   EqualityComparer<TLinkAddress>.Default;
        private static readonly Comparer<TLinkAddress> _comparer =
        ↪   Comparer<TLinkAddress>.Default;
        private readonly IConverter<IList<TLinkAddress>>
        ↪   _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<TLinkAddress> links,
        ↪   IConverter<IList<TLinkAddress>> sequenceToItsLocalElementLevelsConverter) :
        ↪   base(links)
```

```
41            => _sequenceToItsLocalElementLevelsConverter =
    ↪   sequenceToItsLocalElementLevelsConverter;

42

43        /// <summary>
44        /// <para>
45        /// Initializes a new <see cref="OptimalVariantConverter"/> instance.
46        /// </para>
47        /// <para></para>
48        /// </summary>
49        /// <param name="links">
50        /// <para>A links.</para>
51        /// <para></para>
52        /// </param>
53        /// <param name="linkFrequenciesCache">
54        /// <para>A link frequencies cache.</para>
55        /// <para></para>
56        /// </param>
57        [MethodImpl(MethodImplOptions.AggressiveInlining)]
58        public OptimalVariantConverter(ILinks<TLinkAddress> links,
    ↪   LinkFrequenciesCache<TLinkAddress> linkFrequenciesCache)
59            : this(links, new SequenceToItsLocalElementLevelsConverter<TLinkAddress>(links, new
    ↪   FrequenciesCacheBasedLinkToItsFrequencyNumberConverter<TLinkAddress>(linkFrequen⌋
    ↪   ciesCache))) {
    ↪   }

60

61        /// <summary>
62        /// <para>
63        /// Initializes a new <see cref="OptimalVariantConverter"/> instance.
64        /// </para>
65        /// <para></para>
66        /// </summary>
67        /// <param name="links">
68        /// <para>A links.</para>
69        /// <para></para>
70        /// </param>
71        [MethodImpl(MethodImplOptions.AggressiveInlining)]
72        public OptimalVariantConverter(ILinks<TLinkAddress> links)
73            : this(links, new LinkFrequenciesCache<TLinkAddress>(links, new
    ↪   TotalSequenceSymbolFrequencyCounter<TLinkAddress>(links))) { }

74

75        /// <summary>
76        /// <para>
77        /// Converts the sequence.
78        /// </para>
79        /// <para></para>
80        /// </summary>
81        /// <param name="sequence">
82        /// <para>The sequence.</para>
83        /// <para></para>
84        /// </param>
85        /// <returns>
86        /// <para>The link</para>
87        /// <para></para>
88        /// </returns>
89        [MethodImpl(MethodImplOptions.AggressiveInlining)]
90        public override TLinkAddress Convert(IList<TLinkAddress> sequence)
91        {
92            var length = sequence.Count;
93            if (length == 1)
94            {
95                return sequence[0];
96            }
97            if (length == 2)
98            {
99                return _links.GetOrCreate(sequence[0], sequence[1]);
100           }
101           sequence = sequence.ToArray();
102           var levels = _sequenceToItsLocalElementLevelsConverter.Convert(sequence);
103           while (length > 2)
104           {
105               var levelRepeat = 1;
106               var currentLevel = levels[0];
107               var previousLevel = levels[0];
108               var skipOnce = false;
109               var w = 0;
110               for (var i = 1; i < length; i++)
111               {
112                   if (_equalityComparer.Equals(currentLevel, levels[i]))
```

```
113                        {
114                            levelRepeat++;
115                            skipOnce = false;
116                            if (levelRepeat == 2)
117                            {
118                                sequence[w] = _links.GetOrCreate(sequence[i - 1], sequence[i]);
119                                var newLevel = i >= length - 1 ?
120                                    GetPreviousLowerThanCurrentOrCurrent(previousLevel,
                                    ↪  currentLevel) :
121                                    i < 2 ?
122                                    GetNextLowerThanCurrentOrCurrent(currentLevel, levels[i + 1]) :
123                                    GetGreatestNeigbourLowerThanCurrentOrCurrent(previousLevel,
                                    ↪  currentLevel, levels[i + 1]);
124                                levels[w] = newLevel;
125                                previousLevel = currentLevel;
126                                w++;
127                                levelRepeat = 0;
128                                skipOnce = true;
129                            }
130                            else if (i == length - 1)
131                            {
132                                sequence[w] = sequence[i];
133                                levels[w] = levels[i];
134                                w++;
135                            }
136                        }
137                        else
138                        {
139                            currentLevel = levels[i];
140                            levelRepeat = 1;
141                            if (skipOnce)
142                            {
143                                skipOnce = false;
144                            }
145                            else
146                            {
147                                sequence[w] = sequence[i - 1];
148                                levels[w] = levels[i - 1];
149                                previousLevel = levels[w];
150                                w++;
151                            }
152                            if (i == length - 1)
153                            {
154                                sequence[w] = sequence[i];
155                                levels[w] = levels[i];
156                                w++;
157                            }
158                        }
159                    }
160                    length = w;
161                }
162                return _links.GetOrCreate(sequence[0], sequence[1]);
163            }
164            [MethodImpl(MethodImplOptions.AggressiveInlining)]
165            private static TLinkAddress GetGreatestNeigbourLowerThanCurrentOrCurrent(TLinkAddress
                ↪  previous, TLinkAddress current, TLinkAddress next)
166            {
167                return _comparer.Compare(previous, next) > 0
168                    ? _comparer.Compare(previous, current) < 0 ? previous : current
169                    : _comparer.Compare(next, current) < 0 ? next : current;
170            }
171            [MethodImpl(MethodImplOptions.AggressiveInlining)]
172            private static TLinkAddress GetNextLowerThanCurrentOrCurrent(TLinkAddress current,
                ↪  TLinkAddress next) => _comparer.Compare(next, current) < 0 ? next : current;
173            [MethodImpl(MethodImplOptions.AggressiveInlining)]
174            private static TLinkAddress GetPreviousLowerThanCurrentOrCurrent(TLinkAddress previous,
                ↪  TLinkAddress current) => _comparer.Compare(previous, current) < 0 ? previous :
                ↪  current;
175        }
176    }
```

1.5 ./csharp/Platform.Data.Doublets.Sequences/Converters/SequenceToItsLocalElementLevelsConverter.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
```

```csharp
{
    /// <summary>
    /// <para>
    /// Represents the sequence to its local element levels converter.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="LinksOperatorBase{TLinkAddress}"/>
    /// <seealso cref="IConverter{IList{TLinkAddress}}"/>
    public class SequenceToItsLocalElementLevelsConverter<TLinkAddress> :
        LinksOperatorBase<TLinkAddress>, IConverter<IList<TLinkAddress>>
    {
        private static readonly Comparer<TLinkAddress> _comparer =
            Comparer<TLinkAddress>.Default;
        private readonly IConverter<Doublet<TLinkAddress>, TLinkAddress>
            _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<TLinkAddress> links,
            IConverter<Doublet<TLinkAddress>, TLinkAddress> linkToItsFrequencyToNumberConveter)
            : base(links) => _linkToItsFrequencyToNumberConveter =
            linkToItsFrequencyToNumberConveter;

        /// <summary>
        /// <para>
        /// Converts the sequence.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="sequence">
        /// <para>The sequence.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The levels.</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public IList<TLinkAddress> Convert(IList<TLinkAddress> sequence)
        {
            var levels = new TLinkAddress[sequence.Count];
            levels[0] = GetFrequencyNumber(sequence[0], sequence[1]);
            for (var i = 1; i < sequence.Count - 1; i++)
            {
                var previous = GetFrequencyNumber(sequence[i - 1], sequence[i]);
                var next = GetFrequencyNumber(sequence[i], sequence[i + 1]);
                levels[i] = _comparer.Compare(previous, next) > 0 ? previous : next;
            }
            levels[levels.Length - 1] = GetFrequencyNumber(sequence[sequence.Count - 2],
                sequence[sequence.Count - 1]);
            return levels;
        }

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

```
79          /// <para>The target.</para>
80          /// <para></para>
81          /// </param>
82          /// <returns>
83          /// <para>The link</para>
84          /// <para></para>
85          /// </returns>
86          [MethodImpl(MethodImplOptions.AggressiveInlining)]
87          public TLinkAddress GetFrequencyNumber(TLinkAddress source, TLinkAddress target) =>
            ↪  _linkToItsFrequencyToNumberConveter.Convert(new Doublet<TLinkAddress>(source,
            ↪  target));
88      }
89  }
```

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{TLinkAddress}"/>
15     /// <seealso cref="ICriterionMatcher{TLinkAddress}"/>
16     public class DefaultSequenceElementCriterionMatcher<TLinkAddress> :
        ↪  LinksOperatorBase<TLinkAddress>, ICriterionMatcher<TLinkAddress>
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>
26         /// <para></para>
27         /// </param>
28         [MethodImpl(MethodImplOptions.AggressiveInlining)]
29         public DefaultSequenceElementCriterionMatcher(ILinks<TLinkAddress> links) : base(links)
            ↪  { }
30
31         /// <summary>
32         /// <para>
33         /// Determines whether this instance is matched.
34         /// </para>
35         /// <para></para>
36         /// </summary>
37         /// <param name="argument">
38         /// <para>The argument.</para>
39         /// <para></para>
40         /// </param>
41         /// <returns>
42         /// <para>The bool</para>
43         /// <para></para>
44         /// </returns>
45         [MethodImpl(MethodImplOptions.AggressiveInlining)]
46         public bool IsMatched(TLinkAddress argument) => _links.IsPartialPoint(argument);
47     }
48 }
```

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

```
1  using System.Collections.Generic;
2  using System.Runtime.CompilerServices;
3  using Platform.Interfaces;
4
5  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7  namespace Platform.Data.Doublets.Sequences.CriterionMatchers
8  {
9      /// <summary>
10     /// <para>
11     /// Represents the marked sequence criterion matcher.
```

```csharp
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="ICriterionMatcher{TLinkAddress}"/>
    public class MarkedSequenceCriterionMatcher<TLinkAddress> : ICriterionMatcher<TLinkAddress>
    {
        private static readonly EqualityComparer<TLinkAddress> _equalityComparer =
            EqualityComparer<TLinkAddress>.Default;
        private readonly ILinks<TLinkAddress> _links;
        private readonly TLinkAddress _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<TLinkAddress> links, TLinkAddress
            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(TLinkAddress sequenceCandidate)
            => _equalityComparer.Equals(_links.GetSource(sequenceCandidate), _sequenceMarkerLink)
            || !_equalityComparer.Equals(_links.SearchOrDefault(_sequenceMarkerLink,
                sequenceCandidate), _links.Constants.Null);
    }
}
```

1.8 ./csharp/Platform.Data.Doublets.Sequences/CriterionMatchers/UnicodeSequenceMatcher.cs

```csharp
using System.Collections.Generic;
using Platform.Interfaces;

namespace Platform.Data.Doublets.Sequences.CriterionMatchers;

public class UnicodeSequenceMatcher<TLinkAddress> : ICriterionMatcher<TLinkAddress>
{
    public readonly ILinks<TLinkAddress> Storage;
    public readonly TLinkAddress UnicodeSequenceMarker;
    public readonly EqualityComparer<TLinkAddress> EqualityComparer =
        EqualityComparer<TLinkAddress>.Default;
    public UnicodeSequenceMatcher(ILinks<TLinkAddress> storage, TLinkAddress
        unicodeSequenceMarker)
    {
        Storage = storage;
        UnicodeSequenceMarker = unicodeSequenceMarker;
    }
    public bool IsMatched(TLinkAddress argument)
    {
        var target = Storage.GetTarget(argument);
        return EqualityComparer.Equals(UnicodeSequenceMarker, argument) ||
            EqualityComparer.Equals(UnicodeSequenceMarker, target);
    }
}
```

```
21  }

1.9  ./csharp/Platform.Data.Doublets.Sequences/DefaultSequenceAppender.cs
1   using System.Collections.Generic;
2   using System.Runtime.CompilerServices;
3   using Platform.Collections.Stacks;
4   using Platform.Data.Doublets.Sequences.HeightProviders;
5   using Platform.Data.Sequences;
6
7   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
8
9   namespace Platform.Data.Doublets.Sequences
10  {
11      /// <summary>
12      /// <para>
13      /// Represents the default sequence appender.
14      /// </para>
15      /// <para></para>
16      /// </summary>
17      /// <seealso cref="LinksOperatorBase{TLinkAddress}"/>
18      /// <seealso cref="ISequenceAppender{TLinkAddress}"/>
19      public class DefaultSequenceAppender<TLinkAddress> : LinksOperatorBase<TLinkAddress>,
        ↪  ISequenceAppender<TLinkAddress>
20      {
21          private static readonly EqualityComparer<TLinkAddress> _equalityComparer =
            ↪  EqualityComparer<TLinkAddress>.Default;
22          private readonly IStack<TLinkAddress> _stack;
23          private readonly ISequenceHeightProvider<TLinkAddress> _heightProvider;
24
25          /// <summary>
26          /// <para>
27          /// Initializes a new <see cref="DefaultSequenceAppender"/> instance.
28          /// </para>
29          /// <para></para>
30          /// </summary>
31          /// <param name="links">
32          /// <para>A links.</para>
33          /// <para></para>
34          /// </param>
35          /// <param name="stack">
36          /// <para>A stack.</para>
37          /// <para></para>
38          /// </param>
39          /// <param name="heightProvider">
40          /// <para>A height provider.</para>
41          /// <para></para>
42          /// </param>
43          [MethodImpl(MethodImplOptions.AggressiveInlining)]
44          public DefaultSequenceAppender(ILinks<TLinkAddress> links, IStack<TLinkAddress> stack,
            ↪  ISequenceHeightProvider<TLinkAddress> heightProvider)
45              : base(links)
46          {
47              _stack = stack;
48              _heightProvider = heightProvider;
49          }
50
51          /// <summary>
52          /// <para>
53          /// Appends the sequence.
54          /// </para>
55          /// <para></para>
56          /// </summary>
57          /// <param name="sequence">
58          /// <para>The sequence.</para>
59          /// <para></para>
60          /// </param>
61          /// <param name="appendant">
62          /// <para>The appendant.</para>
63          /// <para></para>
64          /// </param>
65          /// <returns>
66          /// <para>The link</para>
67          /// <para></para>
68          /// </returns>
69          [MethodImpl(MethodImplOptions.AggressiveInlining)]
70          public TLinkAddress Append(TLinkAddress sequence, TLinkAddress appendant)
71          {
72              var cursor = sequence;
73              var links = _links;
```

```
74          while (!_equalityComparer.Equals(_heightProvider.Get(cursor), default))
75          {
76              var source = links.GetSource(cursor);
77              var target = links.GetTarget(cursor);
78              if (_equalityComparer.Equals(_heightProvider.Get(source),
   ↪  _heightProvider.Get(target)))
79              {
80                  break;
81              }
82              else
83              {
84                  _stack.Push(source);
85                  cursor = target;
86              }
87          }
88          var left = cursor;
89          var right = appendant;
90          while (!_equalityComparer.Equals(cursor = _stack.PopOrDefault(),
   ↪  links.Constants.Null))
91          {
92              right = links.GetOrCreate(left, right);
93              left = cursor;
94          }
95          return links.GetOrCreate(left, right);
96      }
97  }
98 }
```

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

```
1  using System.Collections.Generic;
2  using System.Linq;
3  using System.Runtime.CompilerServices;
4  using Platform.Interfaces;
5
6  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8  namespace Platform.Data.Doublets.Sequences
9  {
10     /// <summary>
11     /// <para>
12     /// Represents the duplicate segments counter.
13     /// </para>
14     /// <para></para>
15     /// </summary>
16     /// <seealso cref="ICounter{int}"/>
17     public class DuplicateSegmentsCounter<TLinkAddress> : ICounter<int>
18     {
19         private readonly IProvider<IList<KeyValuePair<IList<TLinkAddress>,
   ↪  IList<TLinkAddress>>>> _duplicateFragmentsProvider;
20
21         /// <summary>
22         /// <para>
23         /// Initializes a new <see cref="DuplicateSegmentsCounter"/> instance.
24         /// </para>
25         /// <para></para>
26         /// </summary>
27         /// <param name="duplicateFragmentsProvider">
28         /// <para>A duplicate fragments provider.</para>
29         /// <para></para>
30         /// </param>
31         [MethodImpl(MethodImplOptions.AggressiveInlining)]
32         public DuplicateSegmentsCounter(IProvider<IList<KeyValuePair<IList<TLinkAddress>,
   ↪  IList<TLinkAddress>>>> duplicateFragmentsProvider) => _duplicateFragmentsProvider =
   ↪  duplicateFragmentsProvider;
33
34         /// <summary>
35         /// <para>
36         /// 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 }
```

```csharp
// using System;
// using System.Linq;
// using System.Collections.Generic;
// using System.Runtime.CompilerServices;
// using Platform.Interfaces;
// using Platform.Collections;
// using Platform.Collections.Lists;
// using Platform.Collections.Segments;
// using Platform.Collections.Segments.Walkers;
// using Platform.Singletons;
// using Platform.Converters;
// using Platform.Data.Doublets.Unicode;
//
// #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
//
// namespace Platform.Data.Doublets.Sequences
// {
//     /// <summary>
//     /// <para>
//     /// Represents the duplicate segments provider.
//     /// </para>
//     /// <para></para>
//     /// </summary>
//     /// <seealso cref="DictionaryBasedDuplicateSegmentsWalkerBase{TLinkAddress}"/>
//     /// <seealso cref="IProvider{IList{KeyValuePair{IList{TLinkAddress},
//     IList{TLinkAddress}}}}"/>
//     public class DuplicateSegmentsProvider<TLinkAddress> :
//     DictionaryBasedDuplicateSegmentsWalkerBase<TLinkAddress>,
//     IProvider<IList<KeyValuePair<IList<TLinkAddress>, IList<TLinkAddress>>>>
//     {
//         private static readonly UncheckedConverter<TLinkAddress, long>
//     _addressToInt64Converter = UncheckedConverter<TLinkAddress, long>.Default;
//         private static readonly UncheckedConverter<TLinkAddress, ulong>
//     _addressToUInt64Converter = UncheckedConverter<TLinkAddress, ulong>.Default;
//         private static readonly UncheckedConverter<ulong, TLinkAddress>
//     _uInt64ToAddressConverter = UncheckedConverter<ulong, TLinkAddress>.Default;
//         private readonly ILinks<TLinkAddress> _links;
//         private readonly ILinks<TLinkAddress> _sequences;
//         private HashSet<KeyValuePair<IList<TLinkAddress>, IList<TLinkAddress>>> _groups;
//         private BitString _visited;
//         private class ItemEquilityComparer :
//     IEqualityComparer<KeyValuePair<IList<TLinkAddress>, IList<TLinkAddress>>>
//         {
//             private readonly IListEqualityComparer<TLinkAddress> _listComparer;
//
//             /// <summary>
//             /// <para>
//             /// Initializes a new <see cref="ItemEquilityComparer"/> instance.
//             /// </para>
//             /// <para></para>
//             /// </summary>
//             public ItemEquilityComparer() => _listComparer =
//     Default<IListEqualityComparer<TLinkAddress>>.Instance;
//
//             /// <summary>
//             /// <para>
//             /// Determines whether this instance equals.
//             /// </para>
//             /// <para></para>
//             /// </summary>
//             /// <param name="left">
//             /// <para>The left.</para>
//             /// <para></para>
//             /// </param>
//             /// <param name="right">
//             /// <para>The right.</para>
//             /// <para></para>
//             /// </param>
//             /// <returns>
//             /// <para>The bool</para>
//             /// <para></para>
//             /// </returns>
//             [MethodImpl(MethodImplOptions.AggressiveInlining)]
//             public bool Equals(KeyValuePair<IList<TLinkAddress>, IList<TLinkAddress>> left,
//     KeyValuePair<IList<TLinkAddress>, IList<TLinkAddress>> right) =>
//     _listComparer.Equals(left.Key, right.Key) && _listComparer.Equals(left.Value, right.Value);
```

```
67    //
68    //              /// <summary>
69    //              /// <para>
70    //              /// Gets the hash code using the specified pair.
71    //              /// </para>
72    //              /// <para></para>
73    //              /// </summary>
74    //              /// <param name="pair">
75    //              /// <para>The pair.</para>
76    //              /// <para></para>
77    //              /// </param>
78    //              /// <returns>
79    //              /// <para>The int</para>
80    //              /// <para></para>
81    //              /// </returns>
82    //              [MethodImpl(MethodImplOptions.AggressiveInlining)]
83    //              public int GetHashCode(KeyValuePair<IList<TLinkAddress>, IList<TLinkAddress>>
   ↪  pair) => (_listComparer.GetHashCode(pair.Key),
   ↪  _listComparer.GetHashCode(pair.Value)).GetHashCode();
84    //          }
85    //          private class ItemComparer : IComparer<KeyValuePair<IList<TLinkAddress>,
   ↪  IList<TLinkAddress>>>
86    //          {
87    //              private readonly IListComparer<TLinkAddress> _listComparer;
88    //
89    //              /// <summary>
90    //              /// <para>
91    //              /// Initializes a new <see cref="ItemComparer"/> instance.
92    //              /// </para>
93    //              /// <para></para>
94    //              /// </summary>
95    //              [MethodImpl(MethodImplOptions.AggressiveInlining)]
96    //              public ItemComparer() => _listComparer =
   ↪  Default<IListComparer<TLinkAddress>>.Instance;
97    //
98    //              /// <summary>
99    //              /// <para>
100   //              /// Compares the left.
101   //              /// </para>
102   //              /// <para></para>
103   //              /// </summary>
104   //              /// <param name="left">
105   //              /// <para>The left.</para>
106   //              /// <para></para>
107   //              /// </param>
108   //              /// <param name="right">
109   //              /// <para>The right.</para>
110   //              /// <para></para>
111   //              /// </param>
112   //              /// <returns>
113   //              /// <para>The intermediate result.</para>
114   //              /// <para></para>
115   //              /// </returns>
116   //              [MethodImpl(MethodImplOptions.AggressiveInlining)]
117   //              public int Compare(KeyValuePair<IList<TLinkAddress>, IList<TLinkAddress>> left,
   ↪  KeyValuePair<IList<TLinkAddress>, IList<TLinkAddress>> right)
118   //              {
119   //                  var intermediateResult = _listComparer.Compare(left.Key, right.Key);
120   //                  if (intermediateResult == 0)
121   //                  {
122   //                      intermediateResult = _listComparer.Compare(left.Value, right.Value);
123   //                  }
124   //                  return intermediateResult;
125   //              }
126   //          }
127   //
128   //          /// <summary>
129   //          /// <para>
130   //          /// Initializes a new <see cref="DuplicateSegmentsProvider"/> instance.
131   //          /// </para>
132   //          /// <para></para>
133   //          /// </summary>
134   //          /// <param name="links">
135   //          /// <para>A links.</para>
136   //          /// <para></para>
137   //          /// </param>
138   //          /// <param name="sequences">
```

```
139  //          /// <para>A sequences.</para>
140  //          /// <para></para>
141  //          /// </param>
142  //          [MethodImpl(MethodImplOptions.AggressiveInlining)]
143  //          public DuplicateSegmentsProvider(ILinks<TLinkAddress> links, ILinks<TLinkAddress>
  ↪    sequences)
144  //              : base(minimumStringSegmentLength: 2)
145  //          {
146  //              _links = links;
147  //              _sequences = sequences;
148  //          }
149  //
150  //          /// <summary>
151  //          /// <para>
152  //          /// Gets this instance.
153  //          /// </para>
154  //          /// <para></para>
155  //          /// </summary>
156  //          /// <returns>
157  //          /// <para>The result list.</para>
158  //          /// <para></para>
159  //          /// </returns>
160  //          [MethodImpl(MethodImplOptions.AggressiveInlining)]
161  //          public IList<KeyValuePair<IList<TLinkAddress>, IList<TLinkAddress>>> Get()
162  //          {
163  //              _groups = new HashSet<KeyValuePair<IList<TLinkAddress>,
  ↪   IList<TLinkAddress>>>(Default<ItemEquilityComparer>.Instance);
164  //              var links = _links;
165  //              var count = links.Count();
166  //              _visited = new BitString(_addressToInt64Converter.Convert(count) + 1L);
167  //              links.Each(link =>
168  //              {
169  //                  var linkIndex = links.GetIndex(link);
170  //                  var linkBitIndex = _addressToInt64Converter.Convert(linkIndex);
171  //                  var constants = links.Constants;
172  //                  if (!_visited.Get(linkBitIndex))
173  //                  {
174  //                      var sequenceElements = new List<TLinkAddress>();
175  //                      var filler = new ListFiller<TLinkAddress, TLinkAddress>(sequenceElements,
  ↪   constants.Break);
176  //                      _sequences.Each(filler.AddSkipFirstAndReturnConstant, new
  ↪   LinkAddress<TLinkAddress>(linkIndex));
177  //                      if (sequenceElements.Count > 2)
178  //                      {
179  //                          WalkAll(sequenceElements);
180  //                      }
181  //                  }
182  //                  return constants.Continue;
183  //              });
184  //              var resultList = _groups.ToList();
185  //              var comparer = Default<ItemComparer>.Instance;
186  //              resultList.Sort(comparer);
187  // #if DEBUG
188  //              foreach (var item in resultList)
189  //              {
190  //                  PrintDuplicates(item);
191  //              }
192  // #endif
193  //              return resultList;
194  //          }
195  //
196  //          /// <summary>
197  //          /// <para>
198  //          /// Creates the segment using the specified elements.
199  //          /// </para>
200  //          /// <para></para>
201  //          /// </summary>
202  //          /// <param name="elements">
203  //          /// <para>The elements.</para>
204  //          /// <para></para>
205  //          /// </param>
206  //          /// <param name="offset">
207  //          /// <para>The offset.</para>
208  //          /// <para></para>
209  //          /// </param>
210  //          /// <param name="length">
211  //          /// <para>The length.</para>
```

```
212  //          /// <para></para>
213  //          /// </param>
214  //          /// <returns>
215  //          /// <para>A segment of t link</para>
216  //          /// <para></para>
217  //          /// </returns>
218  //          [MethodImpl(MethodImplOptions.AggressiveInlining)]
219  //          protected override Segment<TLinkAddress> CreateSegment(IList<TLinkAddress> elements,
     ↪   int offset, int length) => new Segment<TLinkAddress>(elements, offset, length);
220  //
221  //          /// <summary>
222  //          /// <para>
223  //          /// Ons the dublicate found using the specified segment.
224  //          /// </para>
225  //          /// <para></para>
226  //          /// </summary>
227  //          /// <param name="segment">
228  //          /// <para>The segment.</para>
229  //          /// <para></para>
230  //          /// </param>
231  //          [MethodImpl(MethodImplOptions.AggressiveInlining)]
232  //          protected override void OnDublicateFound(Segment<TLinkAddress> segment)
233  //          {
234  //              var duplicates = CollectDuplicatesForSegment(segment);
235  //              if (duplicates.Count > 1)
236  //              {
237  //                  _groups.Add(new KeyValuePair<IList<TLinkAddress>,
     ↪   IList<TLinkAddress>>(segment.ToArray(), duplicates));
238  //              }
239  //          }
240  //          [MethodImpl(MethodImplOptions.AggressiveInlining)]
241  //          private List<TLinkAddress> CollectDuplicatesForSegment(Segment<TLinkAddress> segment)
242  //          {
243  //              var duplicates = new List<TLinkAddress>();
244  //              var readAsElement = new HashSet<TLinkAddress>();
245  //              var restrictions = segment.ShiftRight();
246  //              var constants = _links.Constants;
247  //              restrictions[0] = constants.Any;
248  //              _sequences.Each(restrictions, sequence =>
249  //              {
250  //                  var sequenceIndex = sequence[constants.IndexPart];
251  //                  duplicates.Add(sequenceIndex);
252  //                  readAsElement.Add(sequenceIndex);
253  //                  return constants.Continue;
254  //              });
255  //              if (duplicates.Any(x => _visited.Get(_addressToInt64Converter.Convert(x))))
256  //              {
257  //                  return new List<TLinkAddress>();
258  //              }
259  //              foreach (var duplicate in duplicates)
260  //              {
261  //                  var duplicateBitIndex = _addressToInt64Converter.Convert(duplicate);
262  //                  _visited.Set(duplicateBitIndex);
263  //              }
264  //              if (_sequences is Sequences sequencesExperiments)
265  //              {
266  //                  var partiallyMatched = sequencesExperiments.GetAllPartiallyMatchingSequences4
     ↪   ((HashSet<ulong>)(object)readAsElement,
     ↪   (IList<ulong>)segment);
267  //                  foreach (var partiallyMatchedSequence in partiallyMatched)
268  //                  {
269  //                      var sequenceIndex =
     ↪   _uInt64ToAddressConverter.Convert(partiallyMatchedSequence);
270  //                      duplicates.Add(sequenceIndex);
271  //                  }
272  //              }
273  //              duplicates.Sort();
274  //              return duplicates;
275  //          }
276  //          [MethodImpl(MethodImplOptions.AggressiveInlining)]
277  //          private void PrintDuplicates(KeyValuePair<IList<TLinkAddress>, IList<TLinkAddress>>
     ↪   duplicatesItem)
278  //          {
279  //              if (!(_links is ILinks<ulong> ulongLinks))
280  //              {
281  //                  return;
282  //              }
```

```
283  //              var duplicatesKey = duplicatesItem.Key;
284  //              var keyString = UnicodeMap.FromLinksToString((IList<ulong>)duplicatesKey);
285  //              Console.WriteLine($"> {keyString} ({string.Join(", ", duplicatesKey)})");
286  //              var duplicatesList = duplicatesItem.Value;
287  //              for (int i = 0; i < duplicatesList.Count; i++)
288  //              {
289  //                  var sequenceIndex = _addressToUInt64Converter.Convert(duplicatesList[i]);
290  //                  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));
291  //                  Console.WriteLine(formatedSequenceStructure);
292  //                  var sequenceString = UnicodeMap.FromSequenceLinkToString(sequenceIndex,
     ↪  ulongLinks);
293  //                  Console.WriteLine(sequenceString);
294  //              }
295  //              Console.WriteLine();
296  //          }
297  //      }
298  // }
```

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

```csharp
1   using System;
2   using System.Collections.Generic;
3   using System.Runtime.CompilerServices;
4   using Platform.Interfaces;
5   using Platform.Numbers;
6
7   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
8
9   namespace Platform.Data.Doublets.Sequences.Frequencies.Cache
10  {
11      /// <remarks>
12      /// Can be used to operate with many CompressingConverters (to keep global frequencies data
        ↪  between them).
13      /// TODO: Extract interface to implement frequencies storage inside Links storage
14      /// </remarks>
15      public class LinkFrequenciesCache<TLinkAddress> : LinksOperatorBase<TLinkAddress>
16      {
17          private static readonly EqualityComparer<TLinkAddress> _equalityComparer =
            ↪  EqualityComparer<TLinkAddress>.Default;
18          private static readonly Comparer<TLinkAddress> _comparer =
            ↪  Comparer<TLinkAddress>.Default;
19          private static readonly TLinkAddress _zero = default;
20          private static readonly TLinkAddress _one = Arithmetic.Increment(_zero);
21          private readonly Dictionary<Doublet<TLinkAddress>, LinkFrequency<TLinkAddress>>
            ↪  _doubletsCache;
22          private readonly ICounter<TLinkAddress, TLinkAddress> _frequencyCounter;
23
24          /// <summary>
25          /// <para>
26          /// Initializes a new <see cref="LinkFrequenciesCache"/> instance.
27          /// </para>
28          /// <para></para>
29          /// </summary>
30          /// <param name="links">
31          /// <para>A links.</para>
32          /// <para></para>
33          /// </param>
34          /// <param name="frequencyCounter">
35          /// <para>A frequency counter.</para>
36          /// <para></para>
37          /// </param>
38          [MethodImpl(MethodImplOptions.AggressiveInlining)]
39          public LinkFrequenciesCache(ILinks<TLinkAddress> links, ICounter<TLinkAddress,
            ↪  TLinkAddress> frequencyCounter)
40              : base(links)
41          {
42              _doubletsCache = new Dictionary<Doublet<TLinkAddress>,
                ↪  LinkFrequency<TLinkAddress>>(4096, DoubletComparer<TLinkAddress>.Default);
43              _frequencyCounter = frequencyCounter;
44          }
45
46          /// <summary>
47          /// <para>
48          /// Gets the frequency using the specified source.
49          /// </para>
50          /// <para></para>
51          /// </summary>
52          /// <param name="source">
```

```csharp
        /// <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<TLinkAddress> GetFrequency(TLinkAddress source, TLinkAddress target)
        {
            var doublet = new Doublet<TLinkAddress>(source, target);
            return GetFrequency(ref doublet);
        }

        /// <summary>
        /// <para>
        /// Gets 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<TLinkAddress> GetFrequency(ref Doublet<TLinkAddress> doublet)
        {
            _doubletsCache.TryGetValue(doublet, out LinkFrequency<TLinkAddress> data);
            return data;
        }

        /// <summary>
        /// <para>
        /// Increments the frequencies using the specified sequence.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="sequence">
        /// <para>The sequence.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void IncrementFrequencies(IList<TLinkAddress> 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)]
```

```csharp
130         public LinkFrequency<TLinkAddress> IncrementFrequency(TLinkAddress source, TLinkAddress
    ↪  target)
131         {
132             var doublet = new Doublet<TLinkAddress>(source, target);
133             return IncrementFrequency(ref doublet);
134         }
135
136         /// <summary>
137         /// <para>
138         /// Prints the frequencies using the specified sequence.
139         /// </para>
140         /// <para></para>
141         /// </summary>
142         /// <param name="sequence">
143         /// <para>The sequence.</para>
144         /// <para></para>
145         /// </param>
146         [MethodImpl(MethodImplOptions.AggressiveInlining)]
147         public void PrintFrequencies(IList<TLinkAddress> sequence)
148         {
149             for (var i = 1; i < sequence.Count; i++)
150             {
151                 PrintFrequency(sequence[i - 1], sequence[i]);
152             }
153         }
154
155         /// <summary>
156         /// <para>
157         /// Prints the frequency using the specified source.
158         /// </para>
159         /// <para></para>
160         /// </summary>
161         /// <param name="source">
162         /// <para>The source.</para>
163         /// <para></para>
164         /// </param>
165         /// <param name="target">
166         /// <para>The target.</para>
167         /// <para></para>
168         /// </param>
169         [MethodImpl(MethodImplOptions.AggressiveInlining)]
170         public void PrintFrequency(TLinkAddress source, TLinkAddress target)
171         {
172             var number = GetFrequency(source, target).Frequency;
173             Console.WriteLine("({0},{1}) - {2}", source, target, number);
174         }
175
176         /// <summary>
177         /// <para>
178         /// Increments the frequency using the specified doublet.
179         /// </para>
180         /// <para></para>
181         /// </summary>
182         /// <param name="doublet">
183         /// <para>The doublet.</para>
184         /// <para></para>
185         /// </param>
186         /// <returns>
187         /// <para>The data.</para>
188         /// <para></para>
189         /// </returns>
190         [MethodImpl(MethodImplOptions.AggressiveInlining)]
191         public LinkFrequency<TLinkAddress> IncrementFrequency(ref Doublet<TLinkAddress> doublet)
192         {
193             if (_doubletsCache.TryGetValue(doublet, out LinkFrequency<TLinkAddress> data))
194             {
195                 data.IncrementFrequency();
196             }
197             else
198             {
199                 var link = _links.SearchOrDefault(doublet.Source, doublet.Target);
200                 data = new LinkFrequency<TLinkAddress>(_one, link);
201                 if (!_equalityComparer.Equals(link, default))
202                 {
203                     data.Frequency = Arithmetic.Add(data.Frequency,
    ↪  _frequencyCounter.Count(link));
204                 }
205                 _doubletsCache.Add(doublet, data);
```

```
206              }
207              return data;
208          }
209
210          /// <summary>
211          /// <para>
212          /// Validates the frequencies.
213          /// </para>
214          /// <para></para>
215          /// </summary>
216          /// <exception cref="InvalidOperationException">
217          /// <para>Frequencies validation failed.</para>
218          /// <para></para>
219          /// </exception>
220          [MethodImpl(MethodImplOptions.AggressiveInlining)]
221          public void ValidateFrequencies()
222          {
223              foreach (var entry in _doubletsCache)
224              {
225                  var value = entry.Value;
226                  var linkIndex = value.Link;
227                  if (!_equalityComparer.Equals(linkIndex, default))
228                  {
229                      var frequency = value.Frequency;
230                      var count = _frequencyCounter.Count(linkIndex);
231                      // TODO: Why `frequency` always greater than `count` by 1?
232                      if (((_comparer.Compare(frequency, count) > 0) &&
                          ↪  (_comparer.Compare(Arithmetic.Subtract(frequency, count), _one) > 0))
233                          || ((_comparer.Compare(count, frequency) > 0) &&
                          ↪  (_comparer.Compare(Arithmetic.Subtract(count, frequency), _one) > 0)))
234                      {
235                          throw new InvalidOperationException("Frequencies validation failed.");
236                      }
237                  }
238                  //else
239                  //{
240                  //    if (value.Frequency > 0)
241                  //    {
242                  //        var frequency = value.Frequency;
243                  //        linkIndex = _createLink(entry.Key.Source, entry.Key.Target);
244                  //        var count = _countLinkFrequency(linkIndex);
245
246                  //        if ((frequency > count && frequency - count > 1) || (count > frequency
                          ↪  && count - frequency > 1))
247                  //            throw new InvalidOperationException("Frequencies validation
                          ↪  failed.");
248                  //    }
249                  //}
250              }
251          }
252      }
253  }
```

1.13 ./csharp/Platform.Data.Doublets.Sequences/Frequencies/Cache/LinkFrequency.cs

```
1   using System.Runtime.CompilerServices;
2   using Platform.Numbers;
3
4   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6   namespace Platform.Data.Doublets.Sequences.Frequencies.Cache
7   {
8       /// <summary>
9       /// <para>
10      /// Represents the link frequency.
11      /// </para>
12      /// <para></para>
13      /// </summary>
14      public class LinkFrequency<TLinkAddress>
15      {
16          /// <summary>
17          /// <para>
18          /// Gets or sets the frequency value.
19          /// </para>
20          /// <para></para>
21          /// </summary>
22          public TLinkAddress Frequency { get; set; }
23          /// <summary>
24          /// <para>
```

```
25        /// Gets or sets the link value.
26        /// </para>
27        /// <para></para>
28        /// </summary>
29        public TLinkAddress Link { get; set; }
30
31        /// <summary>
32        /// <para>
33        /// Initializes a new <see cref="LinkFrequency"/> instance.
34        /// </para>
35        /// <para></para>
36        /// </summary>
37        /// <param name="frequency">
38        /// <para>A frequency.</para>
39        /// <para></para>
40        /// </param>
41        /// <param name="link">
42        /// <para>A link.</para>
43        /// <para></para>
44        /// </param>
45        [MethodImpl(MethodImplOptions.AggressiveInlining)]
46        public LinkFrequency(TLinkAddress frequency, TLinkAddress link)
47        {
48            Frequency = frequency;
49            Link = link;
50        }
51
52        /// <summary>
53        /// <para>
54        /// Initializes a new <see cref="LinkFrequency"/> instance.
55        /// </para>
56        /// <para></para>
57        /// </summary>
58        [MethodImpl(MethodImplOptions.AggressiveInlining)]
59        public LinkFrequency() { }
60
61        /// <summary>
62        /// <para>
63        /// Increments the frequency.
64        /// </para>
65        /// <para></para>
66        /// </summary>
67        [MethodImpl(MethodImplOptions.AggressiveInlining)]
68        public void IncrementFrequency() => Frequency =
     ↪   Arithmetic<TLinkAddress>.Increment(Frequency);
69
70        /// <summary>
71        /// <para>
72        /// Decrements the frequency.
73        /// </para>
74        /// <para></para>
75        /// </summary>
76        [MethodImpl(MethodImplOptions.AggressiveInlining)]
77        public void DecrementFrequency() => Frequency =
     ↪   Arithmetic<TLinkAddress>.Decrement(Frequency);
78
79        /// <summary>
80        /// <para>
81        /// Returns the string.
82        /// </para>
83        /// <para></para>
84        /// </summary>
85        /// <returns>
86        /// <para>The string</para>
87        /// <para></para>
88        /// </returns>
89        [MethodImpl(MethodImplOptions.AggressiveInlining)]
90        public override string ToString() => $"F: {Frequency}, L: {Link}";
91    }
92 }
```

1.14 ./csharp/Platform.Data.Doublets.Sequences/Frequencies/Cache/LinkToItsFrequencyValueConverter.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.Sequences.Frequencies.Cache
```

```csharp
 7  {
 8      /// <summary>
 9      /// <para>
10      /// Represents the frequencies cache based link to its frequency number converter.
11      /// </para>
12      /// <para></para>
13      /// </summary>
14      /// <seealso cref="IConverter{Doublet{TLinkAddress}, TLinkAddress}"/>
15      public class FrequenciesCacheBasedLinkToItsFrequencyNumberConverter<TLinkAddress> :
    ↪   IConverter<Doublet<TLinkAddress>, TLinkAddress>
16      {
17          private readonly LinkFrequenciesCache<TLinkAddress> _cache;
18
19          /// <summary>
20          /// <para>
21          /// Initializes a new <see
    ↪   cref="FrequenciesCacheBasedLinkToItsFrequencyNumberConverter"/> instance.
22          /// </para>
23          /// <para></para>
24          /// </summary>
25          /// <param name="cache">
26          /// <para>A cache.</para>
27          /// <para></para>
28          /// </param>
29          [MethodImpl(MethodImplOptions.AggressiveInlining)]
30          public FrequenciesCacheBasedLinkToItsFrequencyNumberConverter(LinkFrequenciesCache<TLink
    ↪   Address> cache) => _cache =
    ↪   cache;
31
32          /// <summary>
33          /// <para>
34          /// Converts the source.
35          /// </para>
36          /// <para></para>
37          /// </summary>
38          /// <param name="source">
39          /// <para>The source.</para>
40          /// <para></para>
41          /// </param>
42          /// <returns>
43          /// <para>The link</para>
44          /// <para></para>
45          /// </returns>
46          [MethodImpl(MethodImplOptions.AggressiveInlining)]
47          public TLinkAddress Convert(Doublet<TLinkAddress> source) => _cache.GetFrequency(ref
    ↪   source).Frequency;
48      }
49  }
```

1.15 ./csharp/Platform.Data.Doublets.Sequences/Frequencies/Counters/MarkedSequenceSymbolFrequencyOneOf

```csharp
 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 marked sequence symbol frequency one off counter.
11      /// </para>
12      /// <para></para>
13      /// </summary>
14      /// <seealso cref="SequenceSymbolFrequencyOneOffCounter{TLinkAddress}"/>
15      public class MarkedSequenceSymbolFrequencyOneOffCounter<TLinkAddress> :
    ↪   SequenceSymbolFrequencyOneOffCounter<TLinkAddress>
16      {
17          private readonly ICriterionMatcher<TLinkAddress> _markedSequenceMatcher;
18
19          /// <summary>
20          /// <para>
21          /// Initializes a new <see cref="MarkedSequenceSymbolFrequencyOneOffCounter"/> instance.
22          /// </para>
23          /// <para></para>
24          /// </summary>
25          /// <param name="links">
26          /// <para>A links.</para>
27          /// <para></para>
```

```
28      /// </param>
29      /// <param name="markedSequenceMatcher">
30      /// <para>A marked sequence matcher.</para>
31      /// <para></para>
32      /// </param>
33      /// <param name="sequenceLink">
34      /// <para>A sequence link.</para>
35      /// <para></para>
36      /// </param>
37      /// <param name="symbol">
38      /// <para>A symbol.</para>
39      /// <para></para>
40      /// </param>
41      [MethodImpl(MethodImplOptions.AggressiveInlining)]
42      public MarkedSequenceSymbolFrequencyOneOffCounter(ILinks<TLinkAddress> links,
        ↪  ICriterionMatcher<TLinkAddress> markedSequenceMatcher, TLinkAddress sequenceLink,
        ↪  TLinkAddress symbol)
43          : base(links, sequenceLink, symbol)
44          => _markedSequenceMatcher = markedSequenceMatcher;
45
46      /// <summary>
47      /// <para>
48      /// Counts this instance.
49      /// </para>
50      /// <para></para>
51      /// </summary>
52      /// <returns>
53      /// <para>The link</para>
54      /// <para></para>
55      /// </returns>
56      [MethodImpl(MethodImplOptions.AggressiveInlining)]
57      public override TLinkAddress Count()
58      {
59          if (!_markedSequenceMatcher.IsMatched(_sequenceLink))
60          {
61              return default;
62          }
63          return base.Count();
64      }
65    }
66  }
```

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

```
1   using System.Collections.Generic;
2   using System.Runtime.CompilerServices;
3   using Platform.Interfaces;
4   using Platform.Numbers;
5   using Platform.Data.Sequences;
6
7   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
8
9   namespace Platform.Data.Doublets.Sequences.Frequencies.Counters
10  {
11      /// <summary>
12      /// <para>
13      /// Represents the sequence symbol frequency one off counter.
14      /// </para>
15      /// <para></para>
16      /// </summary>
17      /// <seealso cref="ICounter{TLinkAddress}"/>
18      public class SequenceSymbolFrequencyOneOffCounter<TLinkAddress> : ICounter<TLinkAddress>
19      {
20          private static readonly EqualityComparer<TLinkAddress> _equalityComparer =
            ↪  EqualityComparer<TLinkAddress>.Default;
21          private static readonly Comparer<TLinkAddress> _comparer =
            ↪  Comparer<TLinkAddress>.Default;
22
23          /// <summary>
24          /// <para>
25          /// The links.
26          /// </para>
27          /// <para></para>
28          /// </summary>
29          protected readonly ILinks<TLinkAddress> _links;
30          /// <summary>
31          /// <para>
32          /// The sequence link.
33          /// </para>
34          /// <para></para>
```

```csharp
        /// </summary>
        protected readonly TLinkAddress _sequenceLink;
        /// <summary>
        /// <para>
        /// The symbol.
        /// </para>
        /// <para></para>
        /// </summary>
        protected readonly TLinkAddress _symbol;
        /// <summary>
        /// <para>
        /// The total.
        /// </para>
        /// <para></para>
        /// </summary>
        protected TLinkAddress _total;

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

        /// <summary>
        /// <para>
        /// Counts this instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <returns>
        /// <para>The total.</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public virtual TLinkAddress Count()
        {
            if (_comparer.Compare(_total, default) > 0)
            {
                return _total;
            }
            StopableSequenceWalker.WalkRight(_sequenceLink, _links.GetSource, _links.GetTarget,
                IsElement, VisitElement);
            return _total;
        }
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private bool IsElement(TLinkAddress x) => _equalityComparer.Equals(x, _symbol) ||
            _links.IsPartialPoint(x); // TODO: Use SequenceElementCreteriaMatcher instead of
            IsPartialPoint
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private bool VisitElement(TLinkAddress element)
        {
            if (_equalityComparer.Equals(element, _symbol))
            {
                _total = Arithmetic.Increment(_total);
            }
            return true;
```

```
109            }
110        }
111    }
```

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

```csharp
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{TLinkAddress, TLinkAddress}"/>
15     public class TotalMarkedSequenceSymbolFrequencyCounter<TLinkAddress> :
    ↪    ICounter<TLinkAddress, TLinkAddress>
16     {
17         private readonly ILinks<TLinkAddress> _links;
18         private readonly ICriterionMatcher<TLinkAddress> _markedSequenceMatcher;
19
20         /// <summary>
21         /// <para>
22         /// Initializes a new <see cref="TotalMarkedSequenceSymbolFrequencyCounter"/> instance.
23         /// </para>
24         /// <para></para>
25         /// </summary>
26         /// <param name="links">
27         /// <para>A links.</para>
28         /// <para></para>
29         /// </param>
30         /// <param name="markedSequenceMatcher">
31         /// <para>A marked sequence matcher.</para>
32         /// <para></para>
33         /// </param>
34         [MethodImpl(MethodImplOptions.AggressiveInlining)]
35         public TotalMarkedSequenceSymbolFrequencyCounter(ILinks<TLinkAddress> links,
    ↪    ICriterionMatcher<TLinkAddress> markedSequenceMatcher)
36         {
37             _links = links;
38             _markedSequenceMatcher = markedSequenceMatcher;
39         }
40
41         /// <summary>
42         /// <para>
43         /// Counts the argument.
44         /// </para>
45         /// <para></para>
46         /// </summary>
47         /// <param name="argument">
48         /// <para>The argument.</para>
49         /// <para></para>
50         /// </param>
51         /// <returns>
52         /// <para>The link</para>
53         /// <para></para>
54         /// </returns>
55         [MethodImpl(MethodImplOptions.AggressiveInlining)]
56         public TLinkAddress Count(TLinkAddress argument) => new
    ↪    TotalMarkedSequenceSymbolFrequencyOneOffCounter<TLinkAddress>(_links,
    ↪    _markedSequenceMatcher, argument).Count();
57     }
58  }
```

1.18 ./csharp/Platform.Data.Doublets.Sequences/Frequencies/Counters/TotalMarkedSequenceSymbolFrequencyO

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

```
11      /// Represents the total marked sequence symbol frequency one off counter.
12      /// </para>
13      /// <para></para>
14      /// </summary>
15      /// <seealso cref="TotalSequenceSymbolFrequencyOneOffCounter{TLinkAddress}"/>
16      public class TotalMarkedSequenceSymbolFrequencyOneOffCounter<TLinkAddress> :
   ↪    TotalSequenceSymbolFrequencyOneOffCounter<TLinkAddress>
17      {
18          private readonly ICriterionMatcher<TLinkAddress> _markedSequenceMatcher;
19
20          /// <summary>
21          /// <para>
22          /// Initializes a new <see cref="TotalMarkedSequenceSymbolFrequencyOneOffCounter"/>
   ↪        instance.
23          /// </para>
24          /// <para></para>
25          /// </summary>
26          /// <param name="links">
27          /// <para>A links.</para>
28          /// <para></para>
29          /// </param>
30          /// <param name="markedSequenceMatcher">
31          /// <para>A marked sequence matcher.</para>
32          /// <para></para>
33          /// </param>
34          /// <param name="symbol">
35          /// <para>A symbol.</para>
36          /// <para></para>
37          /// </param>
38          [MethodImpl(MethodImplOptions.AggressiveInlining)]
39          public TotalMarkedSequenceSymbolFrequencyOneOffCounter(ILinks<TLinkAddress> links,
   ↪        ICriterionMatcher<TLinkAddress> markedSequenceMatcher, TLinkAddress symbol)
40              : base(links, symbol)
41              => _markedSequenceMatcher = markedSequenceMatcher;
42
43          /// <summary>
44          /// <para>
45          /// Counts the sequence symbol frequency using the specified link.
46          /// </para>
47          /// <para></para>
48          /// </summary>
49          /// <param name="link">
50          /// <para>The link.</para>
51          /// <para></para>
52          /// </param>
53          [MethodImpl(MethodImplOptions.AggressiveInlining)]
54          protected override void CountSequenceSymbolFrequency(TLinkAddress link)
55          {
56              var symbolFrequencyCounter = new
   ↪            MarkedSequenceSymbolFrequencyOneOffCounter<TLinkAddress>(_links,
   ↪            _markedSequenceMatcher, link, _symbol);
57              _total = Arithmetic.Add(_total, symbolFrequencyCounter.Count());
58          }
59      }
60  }
```

1.19   ./csharp/Platform.Data.Doublets.Sequences/Frequencies/Counters/TotalSequenceSymbolFrequencyCounter.c

```
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 sequence symbol frequency counter.
11      /// </para>
12      /// <para></para>
13      /// </summary>
14      /// <seealso cref="ICounter{TLinkAddress, TLinkAddress}"/>
15      public class TotalSequenceSymbolFrequencyCounter<TLinkAddress> : ICounter<TLinkAddress,
   ↪    TLinkAddress>
16      {
17          private readonly ILinks<TLinkAddress> _links;
18
19          /// <summary>
20          /// <para>
21          /// Initializes a new <see cref="TotalSequenceSymbolFrequencyCounter"/> 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     public TotalSequenceSymbolFrequencyCounter(ILinks<TLinkAddress> links) => _links = links;
31
32     /// <summary>
33     /// <para>
34     /// Counts the symbol.
35     /// </para>
36     /// <para></para>
37     /// </summary>
38     /// <param name="symbol">
39     /// <para>The symbol.</para>
40     /// <para></para>
41     /// </param>
42     /// <returns>
43     /// <para>The link</para>
44     /// <para></para>
45     /// </returns>
46     [MethodImpl(MethodImplOptions.AggressiveInlining)]
47     public TLinkAddress Count(TLinkAddress symbol) => new
       ↪  TotalSequenceSymbolFrequencyOneOffCounter<TLinkAddress>(_links, symbol).Count();
48     }
49  }
```

1.20 ./csharp/Platform.Data.Doublets.Sequences/Frequencies/Counters/TotalSequenceSymbolFrequencyOneOffC

```
1   using System.Collections.Generic;
2   using System.Runtime.CompilerServices;
3   using Platform.Interfaces;
4   using Platform.Numbers;
5
6   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8   namespace Platform.Data.Doublets.Sequences.Frequencies.Counters
9   {
10      /// <summary>
11      /// <para>
12      /// Represents the total sequence symbol frequency one off counter.
13      /// </para>
14      /// <para></para>
15      /// </summary>
16      /// <seealso cref="ICounter{TLinkAddress}"/>
17      public class TotalSequenceSymbolFrequencyOneOffCounter<TLinkAddress> : ICounter<TLinkAddress>
18      {
19          private static readonly EqualityComparer<TLinkAddress> _equalityComparer =
            ↪  EqualityComparer<TLinkAddress>.Default;
20          private static readonly Comparer<TLinkAddress> _comparer =
            ↪  Comparer<TLinkAddress>.Default;
21
22          /// <summary>
23          /// <para>
24          /// The links.
25          /// </para>
26          /// <para></para>
27          /// </summary>
28          protected readonly ILinks<TLinkAddress> _links;
29          /// <summary>
30          /// <para>
31          /// The symbol.
32          /// </para>
33          /// <para></para>
34          /// </summary>
35          protected readonly TLinkAddress _symbol;
36          /// <summary>
37          /// <para>
38          /// The visits.
39          /// </para>
40          /// <para></para>
41          /// </summary>
42          protected readonly HashSet<TLinkAddress> _visits;
43          /// <summary>
44          /// <para>
45          /// The total.
46          /// </para>
```

```csharp
        /// <para></para>
        /// </summary>
        protected TLinkAddress _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<TLinkAddress> links,
        ↪  TLinkAddress symbol)
        {
            _links = links;
            _symbol = symbol;
            _visits = new HashSet<TLinkAddress>();
            _total = default;
        }

        /// <summary>
        /// <para>
        /// Counts this instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <returns>
        /// <para>The total.</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TLinkAddress Count()
        {
            if (_comparer.Compare(_total, default) > 0 || _visits.Count > 0)
            {
                return _total;
            }
            CountCore(_symbol);
            return _total;
        }
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private void CountCore(TLinkAddress link)
        {
            var any = _links.Constants.Any;
            if (_equalityComparer.Equals(_links.Count(any, link), default))
            {
                CountSequenceSymbolFrequency(link);
            }
            else
            {
                _links.Each(EachElementHandler, any, link);
            }
        }

        /// <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 virtual void CountSequenceSymbolFrequency(TLinkAddress link)
        {
            var symbolFrequencyCounter = new
            ↪  SequenceSymbolFrequencyOneOffCounter<TLinkAddress>(_links, link, _symbol);
            _total = Arithmetic.Add(_total, symbolFrequencyCounter.Count());
        }
```

```
123            }
124            [MethodImpl(MethodImplOptions.AggressiveInlining)]
125            private TLinkAddress EachElementHandler(IList<TLinkAddress> doublet)
126            {
127                var constants = _links.Constants;
128                var doubletIndex = doublet[constants.IndexPart];
129                if (_visits.Add(doubletIndex))
130                {
131                    CountCore(doubletIndex);
132                }
133                return constants.Continue;
134            }
135        }
136    }
```

1.21 ./csharp/Platform.Data.Doublets.Sequences/HeightProviders/CachedSequenceHeightProvider.cs

```
1   using System.Collections.Generic;
2   using System.Runtime.CompilerServices;
3   using Platform.Interfaces;
4   using Platform.Converters;
5
6   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8   namespace Platform.Data.Doublets.Sequences.HeightProviders
9   {
10      /// <summary>
11      /// <para>
12      /// Represents the cached sequence height provider.
13      /// </para>
14      /// <para></para>
15      /// </summary>
16      /// <seealso cref="ISequenceHeightProvider{TLinkAddress}"/>
17      public class CachedSequenceHeightProvider<TLinkAddress> :
        ↪  ISequenceHeightProvider<TLinkAddress>
18      {
19          private static readonly EqualityComparer<TLinkAddress> _equalityComparer =
            ↪  EqualityComparer<TLinkAddress>.Default;
20          private readonly TLinkAddress _heightPropertyMarker;
21          private readonly ISequenceHeightProvider<TLinkAddress> _baseHeightProvider;
22          private readonly IConverter<TLinkAddress> _addressToUnaryNumberConverter;
23          private readonly IConverter<TLinkAddress> _unaryNumberToAddressConverter;
24          private readonly IProperties<TLinkAddress, TLinkAddress, TLinkAddress> _propertyOperator;
25
26          /// <summary>
27          /// <para>
28          /// Initializes a new <see cref="CachedSequenceHeightProvider"/> instance.
29          /// </para>
30          /// <para></para>
31          /// </summary>
32          /// <param name="baseHeightProvider">
33          /// <para>A base height provider.</para>
34          /// <para></para>
35          /// </param>
36          /// <param name="addressToUnaryNumberConverter">
37          /// <para>A address to unary number converter.</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          /// <param name="heightPropertyMarker">
45          /// <para>A height property marker.</para>
46          /// <para></para>
47          /// </param>
48          /// <param name="propertyOperator">
49          /// <para>A property operator.</para>
50          /// <para></para>
51          /// </param>
52          [MethodImpl(MethodImplOptions.AggressiveInlining)]
53          public CachedSequenceHeightProvider(
54              ISequenceHeightProvider<TLinkAddress> baseHeightProvider,
55              IConverter<TLinkAddress> addressToUnaryNumberConverter,
56              IConverter<TLinkAddress> unaryNumberToAddressConverter,
57              TLinkAddress heightPropertyMarker,
58              IProperties<TLinkAddress, TLinkAddress, TLinkAddress> propertyOperator)
59          {
60              _heightPropertyMarker = heightPropertyMarker;
61              _baseHeightProvider = baseHeightProvider;
62              _addressToUnaryNumberConverter = addressToUnaryNumberConverter;
```

```
63              _unaryNumberToAddressConverter = unaryNumberToAddressConverter;
64              _propertyOperator = propertyOperator;
65          }
66
67          /// <summary>
68          /// <para>
69          /// Gets the sequence.
70          /// </para>
71          /// <para></para>
72          /// </summary>
73          /// <param name="sequence">
74          /// <para>The sequence.</para>
75          /// <para></para>
76          /// </param>
77          /// <returns>
78          /// <para>The height.</para>
79          /// <para></para>
80          /// </returns>
81          [MethodImpl(MethodImplOptions.AggressiveInlining)]
82          public TLinkAddress Get(TLinkAddress sequence)
83          {
84              TLinkAddress height;
85              var heightValue = _propertyOperator.GetValue(sequence, _heightPropertyMarker);
86              if (_equalityComparer.Equals(heightValue, default))
87              {
88                  height = _baseHeightProvider.Get(sequence);
89                  heightValue = _addressToUnaryNumberConverter.Convert(height);
90                  _propertyOperator.SetValue(sequence, _heightPropertyMarker, heightValue);
91              }
92              else
93              {
94                  height = _unaryNumberToAddressConverter.Convert(heightValue);
95              }
96              return height;
97          }
98      }
99  }
```

1.22 ./csharp/Platform.Data.Doublets.Sequences/HeightProviders/DefaultSequenceRightHeightProvider.cs

```
1   using System.Runtime.CompilerServices;
2   using Platform.Interfaces;
3   using Platform.Numbers;
4
5   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7   namespace Platform.Data.Doublets.Sequences.HeightProviders
8   {
9       /// <summary>
10      /// <para>
11      /// Represents the default sequence right height provider.
12      /// </para>
13      /// <para></para>
14      /// </summary>
15      /// <seealso cref="LinksOperatorBase{TLinkAddress}"/>
16      /// <seealso cref="ISequenceHeightProvider{TLinkAddress}"/>
17      public class DefaultSequenceRightHeightProvider<TLinkAddress> :
        ↪   LinksOperatorBase<TLinkAddress>, ISequenceHeightProvider<TLinkAddress>
18      {
19          private readonly ICriterionMatcher<TLinkAddress> _elementMatcher;
20
21          /// <summary>
22          /// <para>
23          /// Initializes a new <see cref="DefaultSequenceRightHeightProvider"/> instance.
24          /// </para>
25          /// <para></para>
26          /// </summary>
27          /// <param name="links">
28          /// <para>A links.</para>
29          /// <para></para>
30          /// </param>
31          /// <param name="elementMatcher">
32          /// <para>A element matcher.</para>
33          /// <para></para>
34          /// </param>
35          [MethodImpl(MethodImplOptions.AggressiveInlining)]
36          public DefaultSequenceRightHeightProvider(ILinks<TLinkAddress> links,
        ↪   ICriterionMatcher<TLinkAddress> elementMatcher) : base(links) => _elementMatcher =
        ↪   elementMatcher;
37
```

```
38          /// <summary>
39          /// <para>
40          /// Gets the sequence.
41          /// </para>
42          /// <para></para>
43          /// </summary>
44          /// <param name="sequence">
45          /// <para>The sequence.</para>
46          /// <para></para>
47          /// </param>
48          /// <returns>
49          /// <para>The height.</para>
50          /// <para></para>
51          /// </returns>
52          [MethodImpl(MethodImplOptions.AggressiveInlining)]
53          public TLinkAddress Get(TLinkAddress sequence)
54          {
55              var height = default(TLinkAddress);
56              var pairOrElement = sequence;
57              while (!_elementMatcher.IsMatched(pairOrElement))
58              {
59                  pairOrElement = _links.GetTarget(pairOrElement);
60                  height = Arithmetic.Increment(height);
61              }
62              return height;
63          }
64      }
65  }
```

1.23   ./csharp/Platform.Data.Doublets.Sequences/HeightProviders/ISequenceHeightProvider.cs

```
1  using Platform.Interfaces;
2
3  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5  namespace Platform.Data.Doublets.Sequences.HeightProviders
6  {
7      /// <summary>
8      /// <para>
9      /// Defines the sequence height provider.
10     /// </para>
11     /// <para></para>
12     /// </summary>
13     /// <seealso cref="IProvider{TLinkAddress, TLinkAddress}"/>
14     public interface ISequenceHeightProvider<TLinkAddress> : IProvider<TLinkAddress,
    ↪   TLinkAddress>
15     {
16     }
17  }
```

1.24   ./csharp/Platform.Data.Doublets.Sequences/Incrementers/FrequencyIncrementer.cs

```
1  // using System.Collections.Generic;
2  // using System.Runtime.CompilerServices;
3  // using Platform.Incrementers;
4  //
5  // #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6  //
7  // namespace Platform.Data.Doublets.Incrementers
8  // {
9  //      /// <summary>
10 //      /// <para>
11 //      /// Represents the frequency incrementer.
12 //      /// </para>
13 //      /// <para></para>
14 //      /// </summary>
15 //      /// <seealso cref="LinksOperatorBase{TLinkAddress}"/>
16 //      /// <seealso cref="IIncrementer{TLinkAddress}"/>
17 //      public class FrequencyIncrementer<TLinkAddress> : LinksOperatorBase<TLinkAddress>,
    ↪   IIncrementer<TLinkAddress>
18 //      {
19 //          private static readonly EqualityComparer<TLinkAddress> _equalityComparer =
    ↪   EqualityComparer<TLinkAddress>.Default;
20 //          private readonly TLinkAddress _frequencyMarker;
21 //          private readonly TLinkAddress _unaryOne;
22 //          private readonly IIncrementer<TLinkAddress> _unaryNumberIncrementer;
23 //
24 //          /// <summary>
25 //          /// <para>
26 //          /// Initializes a new <see cref="FrequencyIncrementer"/> instance.
```

```
27   //           /// </para>
28   //           /// <para></para>
29   //           /// </summary>
30   //           /// <param name="links">
31   //           /// <para>A links.</para>
32   //           /// <para></para>
33   //           /// </param>
34   //           /// <param name="frequencyMarker">
35   //           /// <para>A frequency marker.</para>
36   //           /// <para></para>
37   //           /// </param>
38   //           /// <param name="unaryOne">
39   //           /// <para>A unary one.</para>
40   //           /// <para></para>
41   //           /// </param>
42   //           /// <param name="unaryNumberIncrementer">
43   //           /// <para>A unary number incrementer.</para>
44   //           /// <para></para>
45   //           /// </param>
46   //           [MethodImpl(MethodImplOptions.AggressiveInlining)]
47   //           public FrequencyIncrementer(ILinks<TLinkAddress> links, TLinkAddress frequencyMarker,
     ↪   TLinkAddress unaryOne, IIncrementer<TLinkAddress> unaryNumberIncrementer)
48   //               : base(links)
49   //           {
50   //               _frequencyMarker = frequencyMarker;
51   //               _unaryOne = unaryOne;
52   //               _unaryNumberIncrementer = unaryNumberIncrementer;
53   //           }
54   //
55   //           /// <summary>
56   //           /// <para>
57   //           /// Increments the frequency.
58   //           /// </para>
59   //           /// <para></para>
60   //           /// </summary>
61   //           /// <param name="frequency">
62   //           /// <para>The frequency.</para>
63   //           /// <para></para>
64   //           /// </param>
65   //           /// <returns>
66   //           /// <para>The link</para>
67   //           /// <para></para>
68   //           /// </returns>
69   //           [MethodImpl(MethodImplOptions.AggressiveInlining)]
70   //           public TLinkAddress Increment(TLinkAddress frequency)
71   //           {
72   //               var links = _links;
73   //               if (_equalityComparer.Equals(frequency, default))
74   //               {
75   //                   return links.GetOrCreate(_unaryOne, _frequencyMarker);
76   //               }
77   //               var incrementedSource =
     ↪   _unaryNumberIncrementer.Increment(links.GetSource(frequency));
78   //               return links.GetOrCreate(incrementedSource, _frequencyMarker);
79   //           }
80   //       }
81   // }
```

1.25 ./csharp/Platform.Data.Doublets.Sequences/Incrementers/UnaryNumberIncrementer.cs

```
1    // using System.Collections.Generic;
2    // using System.Runtime.CompilerServices;
3    // using Platform.Incrementers;
4    //
5    // #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6    //
7    // namespace Platform.Data.Doublets.Incrementers
8    // {
9    //       /// <summary>
10   //       /// <para>
11   //       /// Represents the unary number incrementer.
12   //       /// </para>
13   //       /// <para></para>
14   //       /// </summary>
15   //       /// <seealso cref="LinksOperatorBase{TLinkAddress}"/>
16   //       /// <seealso cref="IIncrementer{TLinkAddress}"/>
17   //       public class UnaryNumberIncrementer<TLinkAddress> : LinksOperatorBase<TLinkAddress>,
     ↪   IIncrementer<TLinkAddress>
18   //       {
```

```csharp
19  //          private static readonly EqualityComparer<TLinkAddress> _equalityComparer =
    ↪   EqualityComparer<TLinkAddress>.Default;
20  //          private readonly TLinkAddress _unaryOne;
21  //
22  //          /// <summary>
23  //          /// <para>
24  //          /// Initializes a new <see cref="UnaryNumberIncrementer"/> instance.
25  //          /// </para>
26  //          /// <para></para>
27  //          /// </summary>
28  //          /// <param name="links">
29  //          /// <para>A links.</para>
30  //          /// <para></para>
31  //          /// </param>
32  //          /// <param name="unaryOne">
33  //          /// <para>A unary one.</para>
34  //          /// <para></para>
35  //          /// </param>
36  //          [MethodImpl(MethodImplOptions.AggressiveInlining)]
37  //          public UnaryNumberIncrementer(ILinks<TLinkAddress> links, TLinkAddress unaryOne) :
    ↪   base(links) => _unaryOne = unaryOne;
38  //
39  //          /// <summary>
40  //          /// <para>
41  //          /// Increments the unary number.
42  //          /// </para>
43  //          /// <para></para>
44  //          /// </summary>
45  //          /// <param name="unaryNumber">
46  //          /// <para>The unary number.</para>
47  //          /// <para></para>
48  //          /// </param>
49  //          /// <returns>
50  //          /// <para>The link</para>
51  //          /// <para></para>
52  //          /// </returns>
53  //          [MethodImpl(MethodImplOptions.AggressiveInlining)]
54  //          public TLinkAddress Increment(TLinkAddress unaryNumber)
55  //          {
56  //              var links = _links;
57  //              if (_equalityComparer.Equals(unaryNumber, _unaryOne))
58  //              {
59  //                  return links.GetOrCreate(_unaryOne, _unaryOne);
60  //              }
61  //              var source = links.GetSource(unaryNumber);
62  //              var target = links.GetTarget(unaryNumber);
63  //              if (_equalityComparer.Equals(source, target))
64  //              {
65  //                  return links.GetOrCreate(unaryNumber, _unaryOne);
66  //              }
67  //              else
68  //              {
69  //                  return links.GetOrCreate(source, Increment(target));
70  //              }
71  //          }
72  //      }
73  // }
```

1.26 ./csharp/Platform.Data.Doublets.Sequences/Indexes/CachedFrequencyIncrementingSequenceIndex.cs

```csharp
1   using System.Collections.Generic;
2   using System.Runtime.CompilerServices;
3   using Platform.Data.Doublets.Sequences.Frequencies.Cache;
4
5   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7   namespace Platform.Data.Doublets.Sequences.Indexes
8   {
9       /// <summary>
10      /// <para>
11      /// Represents the cached frequency incrementing sequence index.
12      /// </para>
13      /// <para></para>
14      /// </summary>
15      /// <seealso cref="ISequenceIndex{TLinkAddress}"/>
16      public class CachedFrequencyIncrementingSequenceIndex<TLinkAddress> :
    ↪   ISequenceIndex<TLinkAddress>
17      {
18          private static readonly EqualityComparer<TLinkAddress> _equalityComparer =
    ↪   EqualityComparer<TLinkAddress>.Default;
```

```csharp
19          private readonly LinkFrequenciesCache<TLinkAddress> _cache;
20
21          /// <summary>
22          /// <para>
23          /// Initializes a new <see cref="CachedFrequencyIncrementingSequenceIndex"/> instance.
24          /// </para>
25          /// <para></para>
26          /// </summary>
27          /// <param name="cache">
28          /// <para>A cache.</para>
29          /// <para></para>
30          /// </param>
31          [MethodImpl(MethodImplOptions.AggressiveInlining)]
32          public CachedFrequencyIncrementingSequenceIndex(LinkFrequenciesCache<TLinkAddress>
    ↪   cache) => _cache = cache;
33
34          /// <summary>
35          /// <para>
36          /// Determines whether this instance add.
37          /// </para>
38          /// <para></para>
39          /// </summary>
40          /// <param name="sequence">
41          /// <para>The sequence.</para>
42          /// <para></para>
43          /// </param>
44          /// <returns>
45          /// <para>The indexed.</para>
46          /// <para></para>
47          /// </returns>
48          [MethodImpl(MethodImplOptions.AggressiveInlining)]
49          public bool Add(IList<TLinkAddress> sequence)
50          {
51              var indexed = true;
52              var i = sequence.Count;
53              while (--i >= 1 && (indexed = IsIndexedWithIncrement(sequence[i - 1], sequence[i])))
    ↪   { }
54              for (; i >= 1; i--)
55              {
56                  _cache.IncrementFrequency(sequence[i - 1], sequence[i]);
57              }
58              return indexed;
59          }
60          [MethodImpl(MethodImplOptions.AggressiveInlining)]
61          private bool IsIndexedWithIncrement(TLinkAddress source, TLinkAddress target)
62          {
63              var frequency = _cache.GetFrequency(source, target);
64              if (frequency == null)
65              {
66                  return false;
67              }
68              var indexed = !_equalityComparer.Equals(frequency.Frequency, default);
69              if (indexed)
70              {
71                  _cache.IncrementFrequency(source, target);
72              }
73              return indexed;
74          }
75
76          /// <summary>
77          /// <para>
78          /// Determines whether this instance might contain.
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 indexed.</para>
88          /// <para></para>
89          /// </returns>
90          [MethodImpl(MethodImplOptions.AggressiveInlining)]
91          public bool MightContain(IList<TLinkAddress> sequence)
92          {
93              var indexed = true;
94              var i = sequence.Count;
```

```
95              while (--i >= 1 && (indexed = IsIndexed(sequence[i - 1], sequence[i]))) { }
96              return indexed;
97          }
98          [MethodImpl(MethodImplOptions.AggressiveInlining)]
99          private bool IsIndexed(TLinkAddress source, TLinkAddress target)
100         {
101             var frequency = _cache.GetFrequency(source, target);
102             if (frequency == null)
103             {
104                 return false;
105             }
106             return !_equalityComparer.Equals(frequency.Frequency, default);
107         }
108     }
109 }
```

1.27 ./csharp/Platform.Data.Doublets.Sequences/Indexes/FrequencyIncrementingSequenceIndex.cs

```
1  // using System.Collections.Generic;
2  // using System.Runtime.CompilerServices;
3  // using Platform.Interfaces;
4  // using Platform.Incrementers;
5  //
6  // #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7  //
8  // namespace Platform.Data.Doublets.Sequences.Indexes
9  // {
10 //     /// <summary>
11 //     /// <para>
12 //     /// Represents the frequency incrementing sequence index.
13 //     /// </para>
14 //     /// <para></para>
15 //     /// </summary>
16 //     /// <seealso cref="SequenceIndex{TLinkAddress}"/>
17 //     /// <seealso cref="ISequenceIndex{TLinkAddress}"/>
18 //     public class FrequencyIncrementingSequenceIndex<TLinkAddress> :
   ↪  SequenceIndex<TLinkAddress>, ISequenceIndex<TLinkAddress>
19 //     {
20 //         private static readonly EqualityComparer<TLinkAddress> _equalityComparer =
   ↪  EqualityComparer<TLinkAddress>.Default;
21 //         private readonly IProperty<TLinkAddress, TLinkAddress> _frequencyPropertyOperator;
22 //         private readonly IIncrementer<TLinkAddress> _frequencyIncrementer;
23 //
24 //         /// <summary>
25 //         /// <para>
26 //         /// Initializes a new <see cref="FrequencyIncrementingSequenceIndex"/> instance.
27 //         /// </para>
28 //         /// <para></para>
29 //         /// </summary>
30 //         /// <param name="links">
31 //         /// <para>A links.</para>
32 //         /// <para></para>
33 //         /// </param>
34 //         /// <param name="frequencyPropertyOperator">
35 //         /// <para>A frequency property operator.</para>
36 //         /// <para></para>
37 //         /// </param>
38 //         /// <param name="frequencyIncrementer">
39 //         /// <para>A frequency incrementer.</para>
40 //         /// <para></para>
41 //         /// </param>
42 //         [MethodImpl(MethodImplOptions.AggressiveInlining)]
43 //         public FrequencyIncrementingSequenceIndex(ILinks<TLinkAddress> links,
   ↪  IProperty<TLinkAddress, TLinkAddress> frequencyPropertyOperator, IIncrementer<TLinkAddress>
   ↪  frequencyIncrementer)
44 //             : base(links)
45 //         {
46 //             _frequencyPropertyOperator = frequencyPropertyOperator;
47 //             _frequencyIncrementer = frequencyIncrementer;
48 //         }
49 //
50 //         /// <summary>
51 //         /// <para>
52 //         /// Determines whether this instance add.
53 //         /// </para>
54 //         /// <para></para>
55 //         /// </summary>
56 //         /// <param name="sequence">
57 //         /// <para>The sequence.</para>
```

```
58  //           /// <para></para>
59  //           /// </param>
60  //           /// <returns>
61  //           /// <para>The indexed.</para>
62  //           /// <para></para>
63  //           /// </returns>
64  //           [MethodImpl(MethodImplOptions.AggressiveInlining)]
65  //           public override bool Add(IList<TLinkAddress> sequence)
66  //           {
67  //               var indexed = true;
68  //               var i = sequence.Count;
69  //               while (--i >= 1 && (indexed = IsIndexedWithIncrement(sequence[i - 1],
↪   sequence[i]))) { }
70  //               for (; i >= 1; i--)
71  //               {
72  //                   Increment(_links.GetOrCreate(sequence[i - 1], sequence[i]));
73  //               }
74  //               return indexed;
75  //           }
76  //           [MethodImpl(MethodImplOptions.AggressiveInlining)]
77  //           private bool IsIndexedWithIncrement(TLinkAddress source, TLinkAddress target)
78  //           {
79  //               var link = _links.SearchOrDefault(source, target);
80  //               var indexed = !_equalityComparer.Equals(link, default);
81  //               if (indexed)
82  //               {
83  //                   Increment(link);
84  //               }
85  //               return indexed;
86  //           }
87  //           [MethodImpl(MethodImplOptions.AggressiveInlining)]
88  //           private void Increment(TLinkAddress link)
89  //           {
90  //               var previousFrequency = _frequencyPropertyOperator.Get(link);
91  //               var frequency = _frequencyIncrementer.Increment(previousFrequency);
92  //               _frequencyPropertyOperator.Set(link, frequency);
93  //           }
94  //       }
95  // }
```

1.28 ./csharp/Platform.Data.Doublets.Sequences/Indexes/ISequenceIndex.cs

```csharp
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<TLinkAddress>
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<TLinkAddress> 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>
```

```csharp
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            bool MightContain(IList<TLinkAddress> sequence);
        }
    }
```

1.29 ./csharp/Platform.Data.Doublets.Sequences/Indexes/SequenceIndex.cs

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

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

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

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

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

        /// <summary>
        /// <para>
        /// Determines whether this instance might contain.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="sequence">
        /// <para>The sequence.</para>
        /// <para></para>
        /// </param>
```

```csharp
        /// <returns>
        /// <para>The indexed.</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public virtual bool MightContain(IList<TLinkAddress> sequence)
        {
            var indexed = true;
            var i = sequence.Count;
            while (--i >= 1 && (indexed =
            ↪   !_equalityComparer.Equals(_links.SearchOrDefault(sequence[i - 1], sequence[i]),
            ↪   default))) { }
            return indexed;
        }
    }
}
```

1.30 ./csharp/Platform.Data.Doublets.Sequences/Indexes/SynchronizedSequenceIndex.cs

```csharp
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{TLinkAddress}"/>
    public class SynchronizedSequenceIndex<TLinkAddress> : ISequenceIndex<TLinkAddress>
    {
        private static readonly EqualityComparer<TLinkAddress> _equalityComparer =
        ↪   EqualityComparer<TLinkAddress>.Default;
        private readonly ISynchronizedLinks<TLinkAddress> _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<TLinkAddress> 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<TLinkAddress> sequence)
        {
            var indexed = true;
            var i = sequence.Count;
            var links = _links.Unsync;
            _links.SyncRoot.DoRead(() =>
            {
                while (--i >= 1 && (indexed =
                ↪   !_equalityComparer.Equals(links.SearchOrDefault(sequence[i - 1],
                ↪   sequence[i]), default))) { }
            });
            if (!indexed)
```

```
58              {
59                  _links.SyncRoot.DoWrite(() =>
60                  {
61                      for (; i >= 1; i--)
62                      {
63                          links.GetOrCreate(sequence[i - 1], sequence[i]);
64                      }
65                  });
66              }
67              return indexed;
68          }
69
70          /// <summary>
71          /// <para>
72          /// Determines whether this instance might contain.
73          /// </para>
74          /// <para></para>
75          /// </summary>
76          /// <param name="sequence">
77          /// <para>The sequence.</para>
78          /// <para></para>
79          /// </param>
80          /// <returns>
81          /// <para>The bool</para>
82          /// <para></para>
83          /// </returns>
84          [MethodImpl(MethodImplOptions.AggressiveInlining)]
85          public bool MightContain(IList<TLinkAddress> sequence)
86          {
87              var links = _links.Unsync;
88              return _links.SyncRoot.DoRead(() =>
89              {
90                  var indexed = true;
91                  var i = sequence.Count;
92                  while (--i >= 1 && (indexed =
                    ↪   !_equalityComparer.Equals(links.SearchOrDefault(sequence[i - 1],
                    ↪   sequence[i]), default))) { }
93                  return indexed;
94              });
95          }
96      }
97  }
```

1.31   ./csharp/Platform.Data.Doublets.Sequences/Indexes/Unindex.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 unindex.
11      /// </para>
12      /// <para></para>
13      /// </summary>
14      /// <seealso cref="ISequenceIndex{TLinkAddress}"/>
15      public class Unindex<TLinkAddress> : ISequenceIndex<TLinkAddress>
16      {
17          /// <summary>
18          /// <para>
19          /// Determines whether this instance add.
20          /// </para>
21          /// <para></para>
22          /// </summary>
23          /// <param name="sequence">
24          /// <para>The sequence.</para>
25          /// <para></para>
26          /// </param>
27          /// <returns>
28          /// <para>The bool</para>
29          /// <para></para>
30          /// </returns>
31          [MethodImpl(MethodImplOptions.AggressiveInlining)]
32          public virtual bool Add(IList<TLinkAddress> sequence) => false;
33
34          /// <summary>
35          /// <para>
```

```
36         /// Determines whether this instance might contain.
37         /// </para>
38         /// <para></para>
39         /// </summary>
40         /// <param name="sequence">
41         /// <para>The sequence.</para>
42         /// <para></para>
43         /// </param>
44         /// <returns>
45         /// <para>The bool</para>
46         /// <para></para>
47         /// </returns>
48         [MethodImpl(MethodImplOptions.AggressiveInlining)]
49         public virtual bool MightContain(IList<TLinkAddress> sequence) => true;
50     }
51 }
```

1.32 ./csharp/Platform.Data.Doublets.Sequences/Numbers/Rational/DecimalToRationalConverter.cs

```
1  using System.Numerics;
2  using Platform.Converters;
3  using Platform.Data.Doublets.Decorators;
4  using System.Globalization;
5  using Platform.Data.Doublets.Numbers.Raw;
6
7  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
8
9  namespace Platform.Data.Doublets.Numbers.Rational
10 {
11     /// <summary>
12     /// <para>
13     /// Represents the decimal to rational converter.
14     /// </para>
15     /// <para></para>
16     /// </summary>
17     /// <seealso cref="LinksDecoratorBase{TLinkAddress}"/>
18     /// <seealso cref="IConverter{decimal, TLinkAddress}"/>
19     public class DecimalToRationalConverter<TLinkAddress> : LinksDecoratorBase<TLinkAddress>,
       ↪   IConverter<decimal, TLinkAddress>
20         where TLinkAddress: struct
21     {
22         /// <summary>
23         /// <para>
24         /// The big integer to raw number sequence converter.
25         /// </para>
26         /// <para></para>
27         /// </summary>
28         public readonly BigIntegerToRawNumberSequenceConverter<TLinkAddress>
           ↪   BigIntegerToRawNumberSequenceConverter;
29
30         /// <summary>
31         /// <para>
32         /// Initializes a new <see cref="DecimalToRationalConverter"/> instance.
33         /// </para>
34         /// <para></para>
35         /// </summary>
36         /// <param name="links">
37         /// <para>A links.</para>
38         /// <para></para>
39         /// </param>
40         /// <param name="bigIntegerToRawNumberSequenceConverter">
41         /// <para>A big integer to raw number sequence converter.</para>
42         /// <para></para>
43         /// </param>
44         public DecimalToRationalConverter(ILinks<TLinkAddress> links,
           ↪   BigIntegerToRawNumberSequenceConverter<TLinkAddress>
           ↪   bigIntegerToRawNumberSequenceConverter) : base(links)
45         {
46             BigIntegerToRawNumberSequenceConverter = bigIntegerToRawNumberSequenceConverter;
47         }
48
49         /// <summary>
50         /// <para>
51         /// Converts the decimal.
52         /// </para>
53         /// <para></para>
54         /// </summary>
55         /// <param name="@decimal">
56         /// <para>The decimal.</para>
57         /// <para></para>
```

```csharp
            /// </param>
            /// <returns>
            /// <para>The link</para>
            /// <para></para>
            /// </returns>
            public TLinkAddress 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);
            }
        }
    }
}
```

1.33 ./csharp/Platform.Data.Doublets.Sequences/Numbers/Rational/RationalToDecimalConverter.cs

```csharp
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{TLinkAddress}"/>
    /// <seealso cref="IConverter{TLinkAddress, decimal}"/>
    public class RationalToDecimalConverter<TLinkAddress> : LinksDecoratorBase<TLinkAddress>,
        IConverter<TLinkAddress, decimal>
        where TLinkAddress: struct
    {
        /// <summary>
        /// <para>
        /// The raw number sequence to big integer converter.
        /// </para>
        /// <para></para>
        /// </summary>
        public readonly RawNumberSequenceToBigIntegerConverter<TLinkAddress>
            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>
```

```
42        public RationalToDecimalConverter(ILinks<TLinkAddress> links,
   ↪    RawNumberSequenceToBigIntegerConverter<TLinkAddress>
   ↪    rawNumberSequenceToBigIntegerConverter) : base(links)
43        {
44            RawNumberSequenceToBigIntegerConverter = rawNumberSequenceToBigIntegerConverter;
45        }
46
47        /// <summary>
48        /// <para>
49        /// Converts the rational number.
50        /// </para>
51        /// <para></para>
52        /// </summary>
53        /// <param name="rationalNumber">
54        /// <para>The rational number.</para>
55        /// <para></para>
56        /// </param>
57        /// <returns>
58        /// <para>The decimal</para>
59        /// <para></para>
60        /// </returns>
61        public decimal Convert(TLinkAddress 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 }
```

1.34   ./csharp/Platform.Data.Doublets.Sequences/Numbers/Raw/BigIntegerToRawNumberSequenceConverter.cs

```
1  using System.Collections.Generic;
2  using System.Numerics;
3  using System.Runtime.InteropServices;
4  using Platform.Converters;
5  using Platform.Data.Doublets.Decorators;
6  using Platform.Numbers;
7  using Platform.Reflection;
8  using Platform.Unsafe;
9
10 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
11
12 namespace Platform.Data.Doublets.Numbers.Raw
13 {
14     /// <summary>
15     /// <para>
16     /// Represents the big integer to raw number sequence converter.
17     /// </para>
18     /// <para></para>
19     /// </summary>
20     /// <seealso cref="LinksDecoratorBase{TLinkAddress}"/>
21     /// <seealso cref="IConverter{BigInteger, TLinkAddress}"/>
22     public class BigIntegerToRawNumberSequenceConverter<TLinkAddress> :
   ↪    LinksDecoratorBase<TLinkAddress>, IConverter<BigInteger, TLinkAddress>
23         where TLinkAddress : struct
24     {
25         /// <summary>
26         /// <para>
27         /// The max value.
28         /// </para>
29         /// <para></para>
30         /// </summary>
31         public static readonly TLinkAddress MaximumValue = NumericType<TLinkAddress>.MaxValue;
32         /// <summary>
33         /// <para>
34         /// The maximum value.
35         /// </para>
36         /// <para></para>
37         /// </summary>
38         public static readonly TLinkAddress BitMask = Bit.ShiftRight(MaximumValue, 1);
39         /// <summary>
40         /// <para>
41         /// The address to number converter.
42         /// </para>
43         /// <para></para>
44         /// </summary>
45         public readonly IConverter<TLinkAddress> AddressToNumberConverter;
```

```csharp
        /// <summary>
        /// <para>
        /// The list to sequence converter.
        /// </para>
        /// <para></para>
        /// </summary>
        public readonly IConverter<IList<TLinkAddress>, TLinkAddress> ListToSequenceConverter;
        /// <summary>
        /// <para>
        /// The negative number marker.
        /// </para>
        /// <para></para>
        /// </summary>
        public readonly TLinkAddress NegativeNumberMarker;

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="BigIntegerToRawNumberSequenceConverter"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="links">
        /// <para>A links.</para>
        /// <para></para>
        /// </param>
        /// <param name="addressToNumberConverter">
        /// <para>A address to number converter.</para>
        /// <para></para>
        /// </param>
        /// <param name="listToSequenceConverter">
        /// <para>A list to sequence converter.</para>
        /// <para></para>
        /// </param>
        /// <param name="negativeNumberMarker">
        /// <para>A negative number marker.</para>
        /// <para></para>
        /// </param>
        public BigIntegerToRawNumberSequenceConverter(ILinks<TLinkAddress> links,
        ↪ IConverter<TLinkAddress> addressToNumberConverter,
        ↪ IConverter<IList<TLinkAddress>,TLinkAddress> listToSequenceConverter, TLinkAddress
        ↪ negativeNumberMarker) : base(links)
        {
            AddressToNumberConverter = addressToNumberConverter;
            ListToSequenceConverter = listToSequenceConverter;
            NegativeNumberMarker = negativeNumberMarker;
        }
        private List<TLinkAddress> GetRawNumberParts(BigInteger bigInteger)
        {
            List<TLinkAddress> rawNumbers = new();
            BigInteger currentBigInt = bigInteger;
            do
            {
                var bigIntBytes = currentBigInt.ToByteArray();
                var bigIntWithBitMask = Bit.And(bigIntBytes.ToStructure<TLinkAddress>(),
                ↪ BitMask);
                var rawNumber = AddressToNumberConverter.Convert(bigIntWithBitMask);
                rawNumbers.Add(rawNumber);
                currentBigInt >>= (NumericType<TLinkAddress>.BitsSize - 1);
            }
            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 TLinkAddress Convert(BigInteger bigInteger)
```

```csharp
                {
                    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;
                }
            }
        }
```

```csharp
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;
```

```csharp
                    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;
                    }
                }
            }
        }
    }
```

1.36 ./csharp/Platform.Data.Doublets.Sequences/Numbers/Raw/NumberToLongRawNumberSequenceConverter.c

```csharp
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.
```

```csharp
        /// </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));
            }
        }
    }
}
```

1.37 ./csharp/Platform.Data.Doublets.Sequences/Numbers/Raw/RawNumberSequenceToBigIntergerConverter.cs

```csharp
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.Reflection;
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{TLinkAddress}"/>
    /// <seealso cref="IConverter{TLinkAddress, BigInteger}"/>
    public class RawNumberSequenceToBigIntegerConverter<TLinkAddress> :
    ↪   LinksDecoratorBase<TLinkAddress>, IConverter<TLinkAddress, BigInteger>
        where TLinkAddress : struct
    {
        /// <summary>
        /// <para>
        /// The default.
        /// </para>
        /// <para></para>
        /// </summary>
        public readonly EqualityComparer<TLinkAddress> EqualityComparer =
        ↪   EqualityComparer<TLinkAddress>.Default;
        /// <summary>
        /// <para>
        /// The number to address converter.
        /// </para>
        /// <para></para>
        /// </summary>
        public readonly IConverter<TLinkAddress, TLinkAddress> NumberToAddressConverter;
        /// <summary>
        /// <para>
        /// The left sequence walker.
        /// </para>
        /// <para></para>
        /// </summary>
```

```csharp
        public readonly LeftSequenceWalker<TLinkAddress> LeftSequenceWalker;
        /// <summary>
        /// <para>
        /// The negative number marker.
        /// </para>
        /// <para></para>
        /// </summary>
        public readonly TLinkAddress 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">
        /// <para>A number to address converter.</para>
        /// <para></para>
        /// </param>
        /// <param name="negativeNumberMarker">
        /// <para>A negative number marker.</para>
        /// <para></para>
        /// </param>
        public RawNumberSequenceToBigIntegerConverter(ILinks<TLinkAddress> links,
        ↪  IConverter<TLinkAddress, TLinkAddress> numberToAddressConverter, TLinkAddress
        ↪  negativeNumberMarker) : base(links)
        {
            NumberToAddressConverter = numberToAddressConverter;
            LeftSequenceWalker = new(links, new DefaultStack<TLinkAddress>());
            NegativeNumberMarker = negativeNumberMarker;
        }

        /// <summary>
        /// <para>
        /// Converts the big integer.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="bigInteger">
        /// <para>The big integer.</para>
        /// <para></para>
        /// </param>
        /// <exception cref="Exception">
        /// <para>Raw number sequence cannot be empty.</para>
        /// <para></para>
        /// </exception>
        /// <returns>
        /// <para>The big integer</para>
        /// <para></para>
        /// </returns>
        public BigInteger Convert(TLinkAddress bigInteger)
        {
            var sign = 1;
            var bigIntegerSequence = bigInteger;
            if (EqualityComparer.Equals(_links.GetSource(bigIntegerSequence),
            ↪  NegativeNumberMarker))
            {
                sign = -1;
                bigIntegerSequence = _links.GetTarget(bigInteger);
            }
            using var enumerator = LeftSequenceWalker.Walk(bigIntegerSequence).GetEnumerator();
            if (!enumerator.MoveNext())
            {
                throw new Exception("Raw number sequence cannot be empty.");
            }
            var nextPart = NumberToAddressConverter.Convert(enumerator.Current);
            BigInteger currentBigInt = new(nextPart.ToBytes());
            while (enumerator.MoveNext())
            {
                currentBigInt <<= (NumericType<TLinkAddress>.BitsSize - 1);
                nextPart = NumberToAddressConverter.Convert(enumerator.Current);
                currentBigInt |= new BigInteger(nextPart.ToBytes());
            }
            return sign == -1 ? BigInteger.Negate(currentBigInt) : currentBigInt;
```

```
121            }
122        }
123    }
```

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

```csharp
                }
                number = Bit.ShiftRight(number, 1);
            }
            return target;
        }
    }
}
```

1.39 ./csharp/Platform.Data.Doublets.Sequences/Numbers/Unary/LinkToItsFrequencyNumberConverter.cs

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

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

namespace Platform.Data.Doublets.Numbers.Unary
{
    /// <summary>
    /// <para>
    /// Represents the link to its frequency number conveter.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="LinksOperatorBase{TLinkAddress}"/>
    /// <seealso cref="IConverter{Doublet{TLinkAddress}, TLinkAddress}"/>
    public class LinkToItsFrequencyNumberConveter<TLinkAddress> :
        LinksOperatorBase<TLinkAddress>, IConverter<Doublet<TLinkAddress>, TLinkAddress>
    {
        private static readonly EqualityComparer<TLinkAddress> _equalityComparer =
            EqualityComparer<TLinkAddress>.Default;
        private readonly IProperty<TLinkAddress, TLinkAddress> _frequencyPropertyOperator;
        private readonly IConverter<TLinkAddress> _unaryNumberToAddressConverter;

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

        /// <summary>
        /// <para>
        /// Converts the doublet.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="doublet">
        /// <para>The doublet.</para>
        /// <para></para>
        /// </param>
        /// <exception cref="ArgumentException">
        /// <para>Link ({doublet}) not found. </para>
        /// <para></para>
        /// </exception>
        /// <returns>
```

```csharp
            /// <para>The link</para>
            /// <para></para>
            /// </returns>
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public TLinkAddress Convert(Doublet<TLinkAddress> doublet)
            {
                var links = _links;
                var link = links.SearchOrDefault(doublet.Source, doublet.Target);
                if (_equalityComparer.Equals(link, default))
                {
                    throw new ArgumentException($"Link ({doublet}) not found.", nameof(doublet));
                }
                var frequency = _frequencyPropertyOperator.Get(link);
                if (_equalityComparer.Equals(frequency, default))
                {
                    return default;
                }
                var frequencyNumber = links.GetSource(frequency);
                return _unaryNumberToAddressConverter.Convert(frequencyNumber);
            }
        }
    }
```

1.40 ./csharp/Platform.Data.Doublets.Sequences/Numbers/Unary/PowerOf2ToUnaryNumberConverter.cs

```csharp
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{TLinkAddress}"/>
    /// <seealso cref="IConverter{int, TLinkAddress}"/>
    public class PowerOf2ToUnaryNumberConverter<TLinkAddress> : LinksOperatorBase<TLinkAddress>,
    ↪  IConverter<int, TLinkAddress>
    {
        private static readonly EqualityComparer<TLinkAddress> _equalityComparer =
        ↪  EqualityComparer<TLinkAddress>.Default;
        private readonly TLinkAddress[] _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<TLinkAddress> links, TLinkAddress one) :
        ↪  base(links)
        {
            _unaryNumberPowersOf2 = new TLinkAddress[64];
            _unaryNumberPowersOf2[0] = one;
        }

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

```
53          /// <para></para>
54          /// </param>
55          /// <returns>
56          /// <para>The power of.</para>
57          /// <para></para>
58          /// </returns>
59          [MethodImpl(MethodImplOptions.AggressiveInlining)]
60          public TLinkAddress Convert(int power)
61          {
62              Ensure.Always.ArgumentInRange(power, new Range<int>(0, _unaryNumberPowersOf2.Length
              ↪  - 1), nameof(power));
63              if (!_equalityComparer.Equals(_unaryNumberPowersOf2[power], default))
64              {
65                  return _unaryNumberPowersOf2[power];
66              }
67              var previousPowerOf2 = Convert(power - 1);
68              var powerOf2 = _links.GetOrCreate(previousPowerOf2, previousPowerOf2);
69              _unaryNumberPowersOf2[power] = powerOf2;
70              return powerOf2;
71          }
72      }
73  }
```

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

```
1   using System.Collections.Generic;
2   using System.Runtime.CompilerServices;
3   using Platform.Converters;
4   using Platform.Numbers;
5
6   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8   namespace Platform.Data.Doublets.Numbers.Unary
9   {
10      /// <summary>
11      /// <para>
12      /// Represents the unary number to address add operation converter.
13      /// </para>
14      /// <para></para>
15      /// </summary>
16      /// <seealso cref="LinksOperatorBase{TLinkAddress}"/>
17      /// <seealso cref="IConverter{TLinkAddress}"/>
18      public class UnaryNumberToAddressAddOperationConverter<TLinkAddress> :
      ↪  LinksOperatorBase<TLinkAddress>, IConverter<TLinkAddress>
19      {
20          private static readonly EqualityComparer<TLinkAddress> _equalityComparer =
          ↪  EqualityComparer<TLinkAddress>.Default;
21          private static readonly UncheckedConverter<TLinkAddress, ulong>
          ↪  _addressToUInt64Converter = UncheckedConverter<TLinkAddress, ulong>.Default;
22          private static readonly UncheckedConverter<ulong, TLinkAddress>
          ↪  _uInt64ToAddressConverter = UncheckedConverter<ulong, TLinkAddress>.Default;
23          private static readonly TLinkAddress _zero = default;
24          private static readonly TLinkAddress _one = Arithmetic.Increment(_zero);
25          private readonly Dictionary<TLinkAddress, TLinkAddress> _unaryToUInt64;
26          private readonly TLinkAddress _unaryOne;
27
28          /// <summary>
29          /// <para>
30          /// Initializes a new <see cref="UnaryNumberToAddressAddOperationConverter"/> instance.
31          /// </para>
32          /// <para></para>
33          /// </summary>
34          /// <param name="links">
35          /// <para>A links.</para>
36          /// <para></para>
37          /// </param>
38          /// <param name="unaryOne">
39          /// <para>A unary one.</para>
40          /// <para></para>
41          /// </param>
42          [MethodImpl(MethodImplOptions.AggressiveInlining)]
43          public UnaryNumberToAddressAddOperationConverter(ILinks<TLinkAddress> links,
          ↪  TLinkAddress unaryOne)
44              : base(links)
45          {
46              _unaryOne = unaryOne;
47              _unaryToUInt64 = CreateUnaryToUInt64Dictionary(links, unaryOne);
48          }
49
50          /// <summary>
```

```csharp
        /// <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 TLinkAddress Convert(TLinkAddress 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];
                TLinkAddress lastValue;
                while (!_unaryToUInt64.TryGetValue(target, out lastValue))
                {
                    source = links.GetSource(target);
                    result = Arithmetic<TLinkAddress>.Add(result, _unaryToUInt64[source]);
                    target = links.GetTarget(target);
                }
                result = Arithmetic<TLinkAddress>.Add(result, lastValue);
                return result;
            }
        }
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static Dictionary<TLinkAddress, TLinkAddress>
        ↪   CreateUnaryToUInt64Dictionary(ILinks<TLinkAddress> links, TLinkAddress unaryOne)
        {
            var unaryToUInt64 = new Dictionary<TLinkAddress, TLinkAddress>
            {
                { 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;
        }
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static TLinkAddress Double(TLinkAddress number) =>
        ↪   _uInt64ToAddressConverter.Convert(_addressToUInt64Converter.Convert(number) * 2UL);
    }
}
```

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

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

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

namespace Platform.Data.Doublets.Numbers.Unary
```

```csharp
{
    /// <summary>
    /// <para>
    /// Represents the unary number to address or operation converter.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="LinksOperatorBase{TLinkAddress}"/>
    /// <seealso cref="IConverter{TLinkAddress}"/>
    public class UnaryNumberToAddressOrOperationConverter<TLinkAddress> :
    ↪    LinksOperatorBase<TLinkAddress>, IConverter<TLinkAddress>
    {
        private static readonly EqualityComparer<TLinkAddress> _equalityComparer =
        ↪    EqualityComparer<TLinkAddress>.Default;
        private static readonly TLinkAddress _zero = default;
        private static readonly TLinkAddress _one = Arithmetic.Increment(_zero);
        private readonly IDictionary<TLinkAddress, int> _unaryNumberPowerOf2Indicies;

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

        /// <summary>
        /// <para>
        /// Converts the source number.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="sourceNumber">
        /// <para>The source number.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The target.</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TLinkAddress Convert(TLinkAddress sourceNumber)
        {
            var links = _links;
            var nullConstant = links.Constants.Null;
            var source = sourceNumber;
            var target = nullConstant;
            if (!_equalityComparer.Equals(source, nullConstant))
            {
                while (true)
                {
                    if (_unaryNumberPowerOf2Indicies.TryGetValue(source, out int powerOf2Index))
                    {
                        SetBit(ref target, powerOf2Index);
                        break;
                    }
                    else
                    {
                        powerOf2Index = _unaryNumberPowerOf2Indicies[links.GetSource(source)];
                        SetBit(ref target, powerOf2Index);
                        source = links.GetTarget(source);
                    }
                }
            }
            return target;
        }
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```csharp
84          private static Dictionary<TLinkAddress, int>
       ↪   CreateUnaryNumberPowerOf2IndiciesDictionary(IConverter<int, TLinkAddress>
       ↪   powerOf2ToUnaryNumberConverter)
85          {
86              var unaryNumberPowerOf2Indicies = new Dictionary<TLinkAddress, int>();
87              for (int i = 0; i < NumericType<TLinkAddress>.BitsSize; i++)
88              {
89                  unaryNumberPowerOf2Indicies.Add(powerOf2ToUnaryNumberConverter.Convert(i), i);
90              }
91              return unaryNumberPowerOf2Indicies;
92          }
93          [MethodImpl(MethodImplOptions.AggressiveInlining)]
94          private static void SetBit(ref TLinkAddress target, int powerOf2Index) => target =
       ↪   Bit.Or(target, Bit.ShiftLeft(_one, powerOf2Index));
95      }
96  }
```

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

```csharp
 1  // using System;
 2  // using System.Collections.Generic;
 3  // using System.Runtime.CompilerServices;
 4  // using System.Linq;
 5  // using System.Text;
 6  // using Platform.Collections;
 7  // using Platform.Collections.Sets;
 8  // using Platform.Collections.Stacks;
 9  // using Platform.Data.Exceptions;
10  // using Platform.Data.Sequences;
11  // using Platform.Data.Doublets.Sequences.Frequencies.Counters;
12  // using Platform.Data.Doublets.Sequences.Walkers;
13  // using LinkIndex = System.UInt64;
14  // using Stack = System.Collections.Generic.Stack<ulong>;
15  //
16  // #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
17  //
18  // namespace Platform.Data.Doublets.Sequences
19  // {
20  //      /// <summary>
21  //      /// <para>
22  //      /// Represents the sequences.
23  //      /// </para>
24  //      /// <para></para>
25  //      /// </summary>
26  //      partial class Sequences
27  //      {
28  //          #region Create All Variants (Not Practical)
29  //
30  //          /// <remarks>
31  //          /// Number of links that is needed to generate all variants for
32  //          /// sequence of length N corresponds to https://oeis.org/A014143/list sequence.
33  //          /// </remarks>
34  //          [MethodImpl(MethodImplOptions.AggressiveInlining)]
35  //          public ulong[] CreateAllVariants2(ulong[] sequence)
36  //          {
37  //              return _sync.DoWrite(() =>
38  //              {
39  //                  if (sequence.IsNullOrEmpty())
40  //                  {
41  //                      return Array.Empty<ulong>();
42  //                  }
43  //                  Links.EnsureLinkExists(sequence);
44  //                  if (sequence.Length == 1)
45  //                  {
46  //                      return sequence;
47  //                  }
48  //                  return CreateAllVariants2Core(sequence, 0, (ulong)sequence.Length - 1);
49  //              });
50  //          }
51  //          [MethodImpl(MethodImplOptions.AggressiveInlining)]
52  //          private ulong[] CreateAllVariants2Core(ulong[] sequence, ulong startAt, ulong stopAt)
53  //          {
54  //              if ((stopAt - startAt) == 0)
55  //              {
56  //                  return new[] { sequence[startAt] };
57  //              }
58  //              if ((stopAt - startAt) == 1)
59  //              {
```

```
60  //                    return new[] { Links.Unsync.GetOrCreate(sequence[startAt], sequence[stopAt])
    ↪  };
61  //                }
62  //                var variants = new ulong[Platform.Numbers.Math.Catalan(stopAt - startAt)];
63  //                var last = 0;
64  //                for (var splitter = startAt; splitter < stopAt; splitter++)
65  //                {
66  //                    var left = CreateAllVariants2Core(sequence, startAt, splitter);
67  //                    var right = CreateAllVariants2Core(sequence, splitter + 1, stopAt);
68  //                    for (var i = 0; i < left.Length; i++)
69  //                    {
70  //                        for (var j = 0; j < right.Length; j++)
71  //                        {
72  //                            var variant = Links.Unsync.GetOrCreate(left[i], right[j]);
73  //                            if (variant == Constants.Null)
74  //                            {
75  //                                throw new NotImplementedException("Creation cancellation is not
    ↪  implemented.");
76  //                            }
77  //                            variants[last++] = variant;
78  //                        }
79  //                    }
80  //                }
81  //                return variants;
82  //            }
83  //
84  //            /// <summary>
85  //            /// <para>
86  //            /// Creates the all variants 1 using the specified sequence.
87  //            /// </para>
88  //            /// <para></para>
89  //            /// </summary>
90  //            /// <param name="sequence">
91  //            /// <para>The sequence.</para>
92  //            /// <para></para>
93  //            /// </param>
94  //            /// <returns>
95  //            /// <para>A list of ulong</para>
96  //            /// <para></para>
97  //            /// </returns>
98  //            [MethodImpl(MethodImplOptions.AggressiveInlining)]
99  //            public List<ulong> CreateAllVariants1(params ulong[] sequence)
100 //            {
101 //                return _sync.DoWrite(() =>
102 //                {
103 //                    if (sequence.IsNullOrEmpty())
104 //                    {
105 //                        return new List<ulong>();
106 //                    }
107 //                    Links.Unsync.EnsureLinkExists(sequence);
108 //                    if (sequence.Length == 1)
109 //                    {
110 //                        return new List<ulong> { sequence[0] };
111 //                    }
112 //                    var results = new
    ↪  List<ulong>((int)Platform.Numbers.Math.Catalan((ulong)sequence.Length));
113 //                    return CreateAllVariants1Core(sequence, results);
114 //                });
115 //            }
116 //            [MethodImpl(MethodImplOptions.AggressiveInlining)]
117 //            private List<ulong> CreateAllVariants1Core(ulong[] sequence, List<ulong> results)
118 //            {
119 //                if (sequence.Length == 2)
120 //                {
121 //                    var link = Links.Unsync.GetOrCreate(sequence[0], sequence[1]);
122 //                    if (link == Constants.Null)
123 //                    {
124 //                        throw new NotImplementedException("Creation cancellation is not
    ↪  implemented.");
125 //                    }
126 //                    results.Add(link);
127 //                    return results;
128 //                }
129 //                var innerSequenceLength = sequence.Length - 1;
130 //                var innerSequence = new ulong[innerSequenceLength];
131 //                for (var li = 0; li < innerSequenceLength; li++)
132 //                {
```

```
133   //                              var link = Links.Unsync.GetOrCreate(sequence[li], sequence[li + 1]);
134   //                              if (link == Constants.Null)
135   //                              {
136   //                                  throw new NotImplementedException("Creation cancellation is not
      ↪   implemented.");
137   //                              }
138   //                              for (var isi = 0; isi < li; isi++)
139   //                              {
140   //                                  innerSequence[isi] = sequence[isi];
141   //                              }
142   //                              innerSequence[li] = link;
143   //                              for (var isi = li + 1; isi < innerSequenceLength; isi++)
144   //                              {
145   //                                  innerSequence[isi] = sequence[isi + 1];
146   //                              }
147   //                              CreateAllVariants1Core(innerSequence, results);
148   //                          }
149   //                          return results;
150   //                      }
151   //
152   //          #endregion
153   //
154   //          /// <summary>
155   //          /// <para>
156   //          /// Eaches the 1 using the specified sequence.
157   //          /// </para>
158   //          /// <para></para>
159   //          /// </summary>
160   //          /// <param name="sequence">
161   //          /// <para>The sequence.</para>
162   //          /// <para></para>
163   //          /// </param>
164   //          /// <returns>
165   //          /// <para>The visited links.</para>
166   //          /// <para></para>
167   //          /// </returns>
168   //          [MethodImpl(MethodImplOptions.AggressiveInlining)]
169   //          public HashSet<ulong> Each1(params ulong[] sequence)
170   //          {
171   //              var visitedLinks = new HashSet<ulong>(); // Заменить на bitstring
172   //              Each1(link =>
173   //              {
174   //                  if (!visitedLinks.Contains(link))
175   //                  {
176   //                      visitedLinks.Add(link); // изучить почему случаются повторы
177   //                  }
178   //                  return true;
179   //              }, sequence);
180   //              return visitedLinks;
181   //          }
182   //          [MethodImpl(MethodImplOptions.AggressiveInlining)]
183   //          private void Each1(Func<ulong, bool> handler, params ulong[] sequence)
184   //          {
185   //              if (sequence.Length == 2)
186   //              {
187   //                  Links.Unsync.Each(sequence[0], sequence[1], handler);
188   //              }
189   //              else
190   //              {
191   //                  var innerSequenceLength = sequence.Length - 1;
192   //                  for (var li = 0; li < innerSequenceLength; li++)
193   //                  {
194   //                      var left = sequence[li];
195   //                      var right = sequence[li + 1];
196   //                      if (left == 0 && right == 0)
197   //                      {
198   //                          continue;
199   //                      }
200   //                      var linkIndex = li;
201   //                      ulong[] innerSequence = null;
202   //                      Links.Unsync.Each(doublet =>
203   //                      {
204   //                          if (innerSequence == null)
205   //                          {
206   //                              innerSequence = new ulong[innerSequenceLength];
207   //                              for (var isi = 0; isi < linkIndex; isi++)
208   //                              {
```

```
209  //                                    innerSequence[isi] = sequence[isi];
210  //                                }
211  //                                for (var isi = linkIndex + 1; isi < innerSequenceLength; isi++)
212  //                                {
213  //                                    innerSequence[isi] = sequence[isi + 1];
214  //                                }
215  //                            }
216  //                            innerSequence[linkIndex] = doublet[Constants.IndexPart];
217  //                            Each1(handler, innerSequence);
218  //                            return Constants.Continue;
219  //                        }, Constants.Any, left, right);
220  //                    }
221  //                }
222  //            }
223  //
224  //            /// <summary>
225  //            /// <para>
226  //            /// Eaches the part using the specified sequence.
227  //            /// </para>
228  //            /// <para></para>
229  //            /// </summary>
230  //            /// <param name="sequence">
231  //            /// <para>The sequence.</para>
232  //            /// <para></para>
233  //            /// </param>
234  //            /// <returns>
235  //            /// <para>The visited links.</para>
236  //            /// <para></para>
237  //            /// </returns>
238  //            [MethodImpl(MethodImplOptions.AggressiveInlining)]
239  //            public HashSet<ulong> EachPart(params ulong[] sequence)
240  //            {
241  //                var visitedLinks = new HashSet<ulong>(); // Заменить на bitstring
242  //                EachPartCore(link =>
243  //                {
244  //                    var linkIndex = link[Constants.IndexPart];
245  //                    if (!visitedLinks.Contains(linkIndex))
246  //                    {
247  //                        visitedLinks.Add(linkIndex); // изучить почему случаются повторы
248  //                    }
249  //                    return Constants.Continue;
250  //                }, sequence);
251  //                return visitedLinks;
252  //            }
253  //
254  //            /// <summary>
255  //            /// <para>
256  //            /// Eaches the part using the specified handler.
257  //            /// </para>
258  //            /// <para></para>
259  //            /// </summary>
260  //            /// <param name="handler">
261  //            /// <para>The handler.</para>
262  //            /// <para></para>
263  //            /// </param>
264  //            /// <param name="sequence">
265  //            /// <para>The sequence.</para>
266  //            /// <para></para>
267  //            /// </param>
268  //            [MethodImpl(MethodImplOptions.AggressiveInlining)]
269  //            public void EachPart(Func<IList<LinkIndex>, LinkIndex> handler, params ulong[]
     ↪  sequence)
270  //            {
271  //                var visitedLinks = new HashSet<ulong>(); // Заменить на bitstring
272  //                EachPartCore(link =>
273  //                {
274  //                    var linkIndex = link[Constants.IndexPart];
275  //                    if (!visitedLinks.Contains(linkIndex))
276  //                    {
277  //                        visitedLinks.Add(linkIndex); // изучить почему случаются повторы
278  //                        return handler(new LinkAddress<LinkIndex>(linkIndex));
279  //                    }
280  //                    return Constants.Continue;
281  //                }, sequence);
282  //            }
283  //            [MethodImpl(MethodImplOptions.AggressiveInlining)]
284  //            private void EachPartCore(Func<IList<LinkIndex>, LinkIndex> handler, params ulong[]
     ↪  sequence)
```

```
285  //            {
286  //                if (sequence.IsNullOrEmpty())
287  //                {
288  //                    return;
289  //                }
290  //                Links.EnsureLinkIsAnyOrExists(sequence);
291  //                if (sequence.Length == 1)
292  //                {
293  //                    var link = sequence[0];
294  //                    if (link > 0)
295  //                    {
296  //                        handler(new LinkAddress<LinkIndex>(link));
297  //                    }
298  //                    else
299  //                    {
300  //                        Links.Each(Constants.Any, Constants.Any, handler);
301  //                    }
302  //                }
303  //                else if (sequence.Length == 2)
304  //                {
305  //                    //_links.Each(sequence[0], sequence[1], handler);
306  //                    //  o_|        x_o ...
307  //                    // x_|          |___|
308  //                    Links.Each(sequence[1], Constants.Any, doublet =>
309  //                    {
310  //                        var match = Links.SearchOrDefault(sequence[0], doublet);
311  //                        if (match != Constants.Null)
312  //                        {
313  //                            handler(new LinkAddress<LinkIndex>(match));
314  //                        }
315  //                        return true;
316  //                    });
317  //                    // |_x        ... x_o
318  //                    // |_o          |___|
319  //                    Links.Each(Constants.Any, sequence[0], doublet =>
320  //                    {
321  //                        var match = Links.SearchOrDefault(doublet, sequence[1]);
322  //                        if (match != 0)
323  //                        {
324  //                            handler(new LinkAddress<LinkIndex>(match));
325  //                        }
326  //                        return true;
327  //                    });
328  //                    //          ._x o_.
329  //                    //            |___|
330  //                    PartialStepRight(x => handler(x), sequence[0], sequence[1]);
331  //                }
332  //                else
333  //                {
334  //                    throw new NotImplementedException();
335  //                }
336  //            }
337  //            [MethodImpl(MethodImplOptions.AggressiveInlining)]
338  //            private void PartialStepRight(Action<IList<LinkIndex>> handler, ulong left, ulong
     ↪  right)
339  //            {
340  //                Links.Unsync.Each(Constants.Any, left, doublet =>
341  //                {
342  //                    StepRight(handler, doublet, right);
343  //                    if (left != doublet)
344  //                    {
345  //                        PartialStepRight(handler, doublet, right);
346  //                    }
347  //                    return true;
348  //                });
349  //            }
350  //            [MethodImpl(MethodImplOptions.AggressiveInlining)]
351  //            private void StepRight(Action<IList<LinkIndex>> handler, ulong left, ulong right)
352  //            {
353  //                Links.Unsync.Each(left, Constants.Any, rightStep =>
354  //                {
355  //                    TryStepRightUp(handler, right, rightStep);
356  //                    return true;
357  //                });
358  //            }
359  //            [MethodImpl(MethodImplOptions.AggressiveInlining)]
360  //            private void TryStepRightUp(Action<IList<LinkIndex>> handler, ulong right, ulong
     ↪  stepFrom)
```

```
361  //              {
362  //                  var upStep = stepFrom;
363  //                  var firstSource = Links.Unsync.GetTarget(upStep);
364  //                  while (firstSource != right && firstSource != upStep)
365  //                  {
366  //                      upStep = firstSource;
367  //                      firstSource = Links.Unsync.GetSource(upStep);
368  //                  }
369  //                  if (firstSource == right)
370  //                  {
371  //                      handler(new LinkAddress<LinkIndex>(stepFrom));
372  //                  }
373  //              }
374  //
375  //              // TODO: Test
376  //              [MethodImpl(MethodImplOptions.AggressiveInlining)]
377  //              private void PartialStepLeft(Action<IList<LinkIndex>> handler, ulong left, ulong
     ↪  right)
378  //              {
379  //                  Links.Unsync.Each(right, Constants.Any, doublet =>
380  //                  {
381  //                      StepLeft(handler, left, doublet);
382  //                      if (right != doublet)
383  //                      {
384  //                          PartialStepLeft(handler, left, doublet);
385  //                      }
386  //                      return true;
387  //                  });
388  //              }
389  //              [MethodImpl(MethodImplOptions.AggressiveInlining)]
390  //              private void StepLeft(Action<IList<LinkIndex>> handler, ulong left, ulong right)
391  //              {
392  //                  Links.Unsync.Each(Constants.Any, right, leftStep =>
393  //                  {
394  //                      TryStepLeftUp(handler, left, leftStep);
395  //                      return true;
396  //                  });
397  //              }
398  //              [MethodImpl(MethodImplOptions.AggressiveInlining)]
399  //              private void TryStepLeftUp(Action<IList<LinkIndex>> handler, ulong left, ulong
     ↪  stepFrom)
400  //              {
401  //                  var upStep = stepFrom;
402  //                  var firstTarget = Links.Unsync.GetSource(upStep);
403  //                  while (firstTarget != left && firstTarget != upStep)
404  //                  {
405  //                      upStep = firstTarget;
406  //                      firstTarget = Links.Unsync.GetTarget(upStep);
407  //                  }
408  //                  if (firstTarget == left)
409  //                  {
410  //                      handler(new LinkAddress<LinkIndex>(stepFrom));
411  //                  }
412  //              }
413  //              [MethodImpl(MethodImplOptions.AggressiveInlining)]
414  //              private bool StartsWith(ulong sequence, ulong link)
415  //              {
416  //                  var upStep = sequence;
417  //                  var firstSource = Links.Unsync.GetSource(upStep);
418  //                  while (firstSource != link && firstSource != upStep)
419  //                  {
420  //                      upStep = firstSource;
421  //                      firstSource = Links.Unsync.GetSource(upStep);
422  //                  }
423  //                  return firstSource == link;
424  //              }
425  //              [MethodImpl(MethodImplOptions.AggressiveInlining)]
426  //              private bool EndsWith(ulong sequence, ulong link)
427  //              {
428  //                  var upStep = sequence;
429  //                  var lastTarget = Links.Unsync.GetTarget(upStep);
430  //                  while (lastTarget != link && lastTarget != upStep)
431  //                  {
432  //                      upStep = lastTarget;
433  //                      lastTarget = Links.Unsync.GetTarget(upStep);
434  //                  }
435  //                  return lastTarget == link;
```

```
436   //              }
437   //
438   //              /// <summary>
439   //              /// <para>
440   //              /// Gets the all matching sequences 0 using the specified sequence.
441   //              /// </para>
442   //              /// <para></para>
443   //              /// </summary>
444   //              /// <param name="sequence">
445   //              /// <para>The sequence.</para>
446   //              /// <para></para>
447   //              /// </param>
448   //              /// <returns>
449   //              /// <para>A list of ulong</para>
450   //              /// <para></para>
451   //              /// </returns>
452   //              [MethodImpl(MethodImplOptions.AggressiveInlining)]
453   //              public List<ulong> GetAllMatchingSequences0(params ulong[] sequence)
454   //              {
455   //                  return _sync.DoRead(() =>
456   //                  {
457   //                      var results = new List<ulong>();
458   //                      if (sequence.Length > 0)
459   //                      {
460   //                          Links.EnsureLinkExists(sequence);
461   //                          var firstElement = sequence[0];
462   //                          if (sequence.Length == 1)
463   //                          {
464   //                              results.Add(firstElement);
465   //                              return results;
466   //                          }
467   //                          if (sequence.Length == 2)
468   //                          {
469   //                              var doublet = Links.SearchOrDefault(firstElement, sequence[1]);
470   //                              if (doublet != Constants.Null)
471   //                              {
472   //                                  results.Add(doublet);
473   //                              }
474   //                              return results;
475   //                          }
476   //                          var linksInSequence = new HashSet<ulong>(sequence);
477   //                          void handler(IList<LinkIndex> result)
478   //                          {
479   //                              var resultIndex = result[Links.Constants.IndexPart];
480   //                              var filterPosition = 0;
481   //                              StopableSequenceWalker.WalkRight(resultIndex, Links.Unsync.GetSource,
      ↪  Links.Unsync.GetTarget,
482   //                                  x => linksInSequence.Contains(x) || Links.Unsync.GetTarget(x) ==
      ↪  x, x =>
483   //                                  {
484   //                                      if (filterPosition == sequence.Length)
485   //                                      {
486   //                                          filterPosition = -2; // Длиннее чем нужно
487   //                                          return false;
488   //                                      }
489   //                                      if (x != sequence[filterPosition])
490   //                                      {
491   //                                          filterPosition = -1;
492   //                                          return false; // Начинается иначе
493   //                                      }
494   //                                      filterPosition++;
495   //
496   //                                      return true;
497   //                                  });
498   //                              if (filterPosition == sequence.Length)
499   //                              {
500   //                                  results.Add(resultIndex);
501   //                              }
502   //                          }
503   //                          if (sequence.Length >= 2)
504   //                          {
505   //                              StepRight(handler, sequence[0], sequence[1]);
506   //                          }
507   //                          var last = sequence.Length - 2;
508   //                          for (var i = 1; i < last; i++)
509   //                          {
510   //                              PartialStepRight(handler, sequence[i], sequence[i + 1]);
511   //                          }
```

```
512    //                        if (sequence.Length >= 3)
513    //                        {
514    //                            StepLeft(handler, sequence[sequence.Length - 2],
       ↪  sequence[sequence.Length - 1]);
515    //                        }
516    //                    }
517    //                    return results;
518    //                });
519    //            }
520    //
521    //            /// <summary>
522    //            /// <para>
523    //            /// Gets the all matching sequences 1 using the specified sequence.
524    //            /// </para>
525    //            /// <para></para>
526    //            /// </summary>
527    //            /// <param name="sequence">
528    //            /// <para>The sequence.</para>
529    //            /// <para></para>
530    //            /// </param>
531    //            /// <returns>
532    //            /// <para>A hash set of ulong</para>
533    //            /// <para></para>
534    //            /// </returns>
535    //            [MethodImpl(MethodImplOptions.AggressiveInlining)]
536    //            public HashSet<ulong> GetAllMatchingSequences1(params ulong[] sequence)
537    //            {
538    //                return _sync.DoRead(() =>
539    //                {
540    //                    var results = new HashSet<ulong>();
541    //                    if (sequence.Length > 0)
542    //                    {
543    //                        Links.EnsureLinkExists(sequence);
544    //                        var firstElement = sequence[0];
545    //                        if (sequence.Length == 1)
546    //                        {
547    //                            results.Add(firstElement);
548    //                            return results;
549    //                        }
550    //                        if (sequence.Length == 2)
551    //                        {
552    //                            var doublet = Links.SearchOrDefault(firstElement, sequence[1]);
553    //                            if (doublet != Constants.Null)
554    //                            {
555    //                                results.Add(doublet);
556    //                            }
557    //                            return results;
558    //                        }
559    //                        var matcher = new Matcher(this, sequence, results, null);
560    //                        if (sequence.Length >= 2)
561    //                        {
562    //                            StepRight(matcher.AddFullMatchedToResults, sequence[0], sequence[1]);
563    //                        }
564    //                        var last = sequence.Length - 2;
565    //                        for (var i = 1; i < last; i++)
566    //                        {
567    //                            PartialStepRight(matcher.AddFullMatchedToResults, sequence[i],
       ↪  sequence[i + 1]);
568    //                        }
569    //                        if (sequence.Length >= 3)
570    //                        {
571    //                            StepLeft(matcher.AddFullMatchedToResults, sequence[sequence.Length -
       ↪  2], sequence[sequence.Length - 1]);
572    //                        }
573    //                    }
574    //                    return results;
575    //                });
576    //            }
577    //
578    //            /// <summary>
579    //            /// <para>
580    //            /// The max sequence format size.
581    //            /// </para>
582    //            /// <para></para>
583    //            /// </summary>
584    //            public const int MaxSequenceFormatSize = 200;
585    //
```

```
586  //           /// <summary>
587  //           /// <para>
588  //           /// Formats the sequence using the specified sequence link.
589  //           /// </para>
590  //           /// <para></para>
591  //           /// </summary>
592  //           /// <param name="sequenceLink">
593  //           /// <para>The sequence link.</para>
594  //           /// <para></para>
595  //           /// </param>
596  //           /// <param name="knownElements">
597  //           /// <para>The known elements.</para>
598  //           /// <para></para>
599  //           /// </param>
600  //           /// <returns>
601  //           /// <para>The string</para>
602  //           /// <para></para>
603  //           /// </returns>
604  //           [MethodImpl(MethodImplOptions.AggressiveInlining)]
605  //           public string FormatSequence(LinkIndex sequenceLink, params LinkIndex[]
   ↪  knownElements) => FormatSequence(sequenceLink, (sb, x) => sb.Append(x), true, knownElements);
606  //
607  //           /// <summary>
608  //           /// <para>
609  //           /// Formats the sequence using the specified sequence link.
610  //           /// </para>
611  //           /// <para></para>
612  //           /// </summary>
613  //           /// <param name="sequenceLink">
614  //           /// <para>The sequence link.</para>
615  //           /// <para></para>
616  //           /// </param>
617  //           /// <param name="elementToString">
618  //           /// <para>The element to string.</para>
619  //           /// <para></para>
620  //           /// </param>
621  //           /// <param name="insertComma">
622  //           /// <para>The insert comma.</para>
623  //           /// <para></para>
624  //           /// </param>
625  //           /// <param name="knownElements">
626  //           /// <para>The known elements.</para>
627  //           /// <para></para>
628  //           /// </param>
629  //           /// <returns>
630  //           /// <para>The string</para>
631  //           /// <para></para>
632  //           /// </returns>
633  //           [MethodImpl(MethodImplOptions.AggressiveInlining)]
634  //           public string FormatSequence(LinkIndex sequenceLink, Action<StringBuilder, LinkIndex>
   ↪  elementToString, bool insertComma, params LinkIndex[] knownElements) =>
   ↪  Links.SyncRoot.DoRead(() => FormatSequence(Links.Unsync, sequenceLink, elementToString,
   ↪  insertComma, knownElements));
635  //           [MethodImpl(MethodImplOptions.AggressiveInlining)]
636  //           private string FormatSequence(ILinks<LinkIndex> links, LinkIndex sequenceLink,
   ↪  Action<StringBuilder, LinkIndex> elementToString, bool insertComma, params LinkIndex[]
   ↪  knownElements)
637  //           {
638  //               var linksInSequence = new HashSet<ulong>(knownElements);
639  //               //var entered = new HashSet<ulong>();
640  //               var sb = new StringBuilder();
641  //               sb.Append('{');
642  //               if (links.Exists(sequenceLink))
643  //               {
644  //                   StopableSequenceWalker.WalkRight(sequenceLink, links.GetSource,
   ↪  links.GetTarget,
645  //                       x => linksInSequence.Contains(x) || links.IsPartialPoint(x), element =>
   ↪  // entered.AddAndReturnVoid, x => { }, entered.DoNotContains
646  //                       {
647  //                           if (insertComma && sb.Length > 1)
648  //                           {
649  //                               sb.Append(',');
650  //                           }
651  //                           //if (entered.Contains(element))
652  //                           //{
653  //                           //    sb.Append('{');
654  //                           //    elementToString(sb, element);
```

```
655   //                          //    sb.Append('}');
656   //                          //}
657   //                          //else
658   //                          elementToString(sb, element);
659   //                          if (sb.Length < MaxSequenceFormatSize)
660   //                          {
661   //                              return true;
662   //                          }
663   //                          sb.Append(insertComma ? ", ..." : "...");
664   //                          return false;
665   //                      });
666   //                  }
667   //                  sb.Append('}');
668   //                  return sb.ToString();
669   //              }
670   //
671   //          /// <summary>
672   //          /// <para>
673   //          /// Safes the format sequence using the specified sequence link.
674   //          /// </para>
675   //          /// <para></para>
676   //          /// </summary>
677   //          /// <param name="sequenceLink">
678   //          /// <para>The sequence link.</para>
679   //          /// <para></para>
680   //          /// </param>
681   //          /// <param name="knownElements">
682   //          /// <para>The known elements.</para>
683   //          /// <para></para>
684   //          /// </param>
685   //          /// <returns>
686   //          /// <para>The string</para>
687   //          /// <para></para>
688   //          /// </returns>
689   //          [MethodImpl(MethodImplOptions.AggressiveInlining)]
690   //          public string SafeFormatSequence(LinkIndex sequenceLink, params LinkIndex[]
      ↪  knownElements) => SafeFormatSequence(sequenceLink, (sb, x) => sb.Append(x), true,
      ↪  knownElements);
691   //
692   //          /// <summary>
693   //          /// <para>
694   //          /// Safes the format sequence using the specified sequence link.
695   //          /// </para>
696   //          /// <para></para>
697   //          /// </summary>
698   //          /// <param name="sequenceLink">
699   //          /// <para>The sequence link.</para>
700   //          /// <para></para>
701   //          /// </param>
702   //          /// <param name="elementToString">
703   //          /// <para>The element to string.</para>
704   //          /// <para></para>
705   //          /// </param>
706   //          /// <param name="insertComma">
707   //          /// <para>The insert comma.</para>
708   //          /// <para></para>
709   //          /// </param>
710   //          /// <param name="knownElements">
711   //          /// <para>The known elements.</para>
712   //          /// <para></para>
713   //          /// </param>
714   //          /// <returns>
715   //          /// <para>The string</para>
716   //          /// <para></para>
717   //          /// </returns>
718   //          [MethodImpl(MethodImplOptions.AggressiveInlining)]
719   //          public string SafeFormatSequence(LinkIndex sequenceLink, Action<StringBuilder,
      ↪  LinkIndex> elementToString, bool insertComma, params LinkIndex[] knownElements) =>
      ↪  Links.SyncRoot.DoRead(() => SafeFormatSequence(Links.Unsync, sequenceLink, elementToString,
      ↪  insertComma, knownElements));
720   //          [MethodImpl(MethodImplOptions.AggressiveInlining)]
721   //          private string SafeFormatSequence(ILinks<LinkIndex> links, LinkIndex sequenceLink,
      ↪  Action<StringBuilder, LinkIndex> elementToString, bool insertComma, params LinkIndex[]
      ↪  knownElements)
722   //          {
723   //              var linksInSequence = new HashSet<ulong>(knownElements);
724   //              var entered = new HashSet<ulong>();
```

```
725  //                var sb = new StringBuilder();
726  //                sb.Append('{');
727  //                if (links.Exists(sequenceLink))
728  //                {
729  //                    StopableSequenceWalker.WalkRight(sequenceLink, links.GetSource,
     ↪   links.GetTarget,
730  //                        x => linksInSequence.Contains(x) || links.IsFullPoint(x),
     ↪   entered.AddAndReturnVoid, x => { }, entered.DoNotContains, element =>
731  //                    {
732  //                        if (insertComma && sb.Length > 1)
733  //                        {
734  //                            sb.Append(',');
735  //                        }
736  //                        if (entered.Contains(element))
737  //                        {
738  //                            sb.Append('{');
739  //                            elementToString(sb, element);
740  //                            sb.Append('}');
741  //                        }
742  //                        else
743  //                        {
744  //                            elementToString(sb, element);
745  //                        }
746  //                        if (sb.Length < MaxSequenceFormatSize)
747  //                        {
748  //                            return true;
749  //                        }
750  //                        sb.Append(insertComma ? ", ..." : "...");
751  //                        return false;
752  //                    });
753  //                }
754  //                sb.Append('}');
755  //                return sb.ToString();
756  //            }
757  //
758  //            /// <summary>
759  //            /// <para>
760  //            /// Gets the all partially matching sequences 0 using the specified sequence.
761  //            /// </para>
762  //            /// <para></para>
763  //            /// </summary>
764  //            /// <param name="sequence">
765  //            /// <para>The sequence.</para>
766  //            /// <para></para>
767  //            /// </param>
768  //            /// <returns>
769  //            /// <para>A list of ulong</para>
770  //            /// <para></para>
771  //            /// </returns>
772  //            [MethodImpl(MethodImplOptions.AggressiveInlining)]
773  //            public List<ulong> GetAllPartiallyMatchingSequences0(params ulong[] sequence)
774  //            {
775  //                return _sync.DoRead(() =>
776  //                {
777  //                    if (sequence.Length > 0)
778  //                    {
779  //                        Links.EnsureLinkExists(sequence);
780  //                        var results = new HashSet<ulong>();
781  //                        for (var i = 0; i < sequence.Length; i++)
782  //                        {
783  //                            AllUsagesCore(sequence[i], results);
784  //                        }
785  //                        var filteredResults = new List<ulong>();
786  //                        var linksInSequence = new HashSet<ulong>(sequence);
787  //                        foreach (var result in results)
788  //                        {
789  //                            var filterPosition = -1;
790  //                            StopableSequenceWalker.WalkRight(result, Links.Unsync.GetSource,
     ↪   Links.Unsync.GetTarget,
791  //                                x => linksInSequence.Contains(x) || Links.Unsync.GetTarget(x) ==
     ↪   x, x =>
792  //                                {
793  //                                    if (filterPosition == (sequence.Length - 1))
794  //                                    {
795  //                                        return false;
796  //                                    }
797  //                                    if (filterPosition >= 0)
798  //                                    {
```

```
//                                        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.DoRead(() =>
//            {
//                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>
```

```
877   //          /// </param>
878   //          /// <returns>
879   //          /// <para>The bool</para>
880   //          /// <para></para>
881   //          /// </returns>
882   //          [MethodImpl(MethodImplOptions.AggressiveInlining)]
883   //          public bool GetAllPartiallyMatchingSequences2(Func<IList<LinkIndex>, LinkIndex>
      ↪  handler, params ulong[] sequence)
884   //          {
885   //              return _sync.DoRead(() =>
886   //              {
887   //                  if (sequence.Length > 0)
888   //                  {
889   //                      Links.EnsureLinkExists(sequence);
890   //
891   //                      var results = new HashSet<ulong>();
892   //                      var filteredResults = new HashSet<ulong>();
893   //                      var matcher = new Matcher(this, sequence, filteredResults, handler);
894   //                      for (var i = 0; i < sequence.Length; i++)
895   //                      {
896   //                          if (!AllUsagesCore1(sequence[i], results,
      ↪  matcher.HandlePartialMatched))
897   //                          {
898   //                              return false;
899   //                          }
900   //                      }
901   //                      return true;
902   //                  }
903   //                  return true;
904   //              });
905   //          }
906   //
907   //          //public HashSet<ulong> GetAllPartiallyMatchingSequences3(params ulong[] sequence)
908   //          //{
909   //          //    return Sync.DoRead(() =>
910   //          //    {
911   //          //        if (sequence.Length > 0)
912   //          //        {
913   //          //            _links.EnsureEachLinkIsAnyOrExists(sequence);
914   //
915   //          //            var firstResults = new HashSet<ulong>();
916   //          //            var lastResults = new HashSet<ulong>();
917   //
918   //          //            var first = sequence.First(x => x != LinksConstants.Any);
919   //          //            var last = sequence.Last(x => x != LinksConstants.Any);
920   //
921   //          //            AllUsagesCore(first, firstResults);
922   //          //            AllUsagesCore(last, lastResults);
923   //
924   //          //            firstResults.IntersectWith(lastResults);
925   //
926   //          //            //for (var i = 0; i < sequence.Length; i++)
927   //          //            //    AllUsagesCore(sequence[i], results);
928   //
929   //          //            var filteredResults = new HashSet<ulong>();
930   //          //            var matcher = new Matcher(this, sequence, filteredResults, null);
931   //          //            matcher.AddAllPartialMatchedToResults(firstResults);
932   //          //            return filteredResults;
933   //          //        }
934   //
935   //          //        return new HashSet<ulong>();
936   //          //    });
937   //          //}
938   //
939   //          /// <summary>
940   //          /// <para>
941   //          /// Gets the all partially matching sequences 3 using the specified sequence.
942   //          /// </para>
943   //          /// <para></para>
944   //          /// </summary>
945   //          /// <param name="sequence">
946   //          /// <para>The sequence.</para>
947   //          /// <para></para>
948   //          /// </param>
949   //          /// <returns>
950   //          /// <para>A hash set of ulong</para>
951   //          /// <para></para>
```

```
952  //            /// </returns>
953  //            [MethodImpl(MethodImplOptions.AggressiveInlining)]
954  //            public HashSet<ulong> GetAllPartiallyMatchingSequences3(params ulong[] sequence)
955  //            {
956  //                return _sync.DoRead(() =>
957  //                {
958  //                    if (sequence.Length > 0)
959  //                    {
960  //                        ILinksExtensions.EnsureLinkIsAnyOrExists(Links, sequence);
961  //                        var firstResults = new HashSet<ulong>();
962  //                        var lastResults = new HashSet<ulong>();
963  //                        var first = sequence.First(x => x != Constants.Any);
964  //                        var last = sequence.Last(x => x != Constants.Any);
965  //                        AllUsagesCore(first, firstResults);
966  //                        AllUsagesCore(last, lastResults);
967  //                        firstResults.IntersectWith(lastResults);
968  //                        //for (var i = 0; i < sequence.Length; i++)
969  //                        //    AllUsagesCore(sequence[i], results);
970  //                        var filteredResults = new HashSet<ulong>();
971  //                        var matcher = new Matcher(this, sequence, filteredResults, null);
972  //                        matcher.AddAllPartialMatchedToResults(firstResults);
973  //                        return filteredResults;
974  //                    }
975  //                    return new HashSet<ulong>();
976  //                });
977  //            }
978  //
979  //            /// <summary>
980  //            /// <para>
981  //            /// Gets the all partially matching sequences 4 using the specified read as elements.
982  //            /// </para>
983  //            /// <para></para>
984  //            /// </summary>
985  //            /// <param name="readAsElements">
986  //            /// <para>The read as elements.</para>
987  //            /// <para></para>
988  //            /// </param>
989  //            /// <param name="sequence">
990  //            /// <para>The sequence.</para>
991  //            /// <para></para>
992  //            /// </param>
993  //            /// <returns>
994  //            /// <para>A hash set of ulong</para>
995  //            /// <para></para>
996  //            /// </returns>
997  //            [MethodImpl(MethodImplOptions.AggressiveInlining)]
998  //            public HashSet<ulong> GetAllPartiallyMatchingSequences4(HashSet<ulong>
     ↪  readAsElements, IList<ulong> sequence)
999  //            {
1000 //                return _sync.DoRead(() =>
1001 //                {
1002 //                    if (sequence.Count > 0)
1003 //                    {
1004 //                        Links.EnsureLinkExists(sequence);
1005 //                        var results = new HashSet<LinkIndex>();
1006 //                        //var nextResults = new HashSet<ulong>();
1007 //                        //for (var i = 0; i < sequence.Length; i++)
1008 //                        //{
1009 //                        //    AllUsagesCore(sequence[i], nextResults);
1010 //                        //    if (results.IsNullOrEmpty())
1011 //                        //    {
1012 //                        //        results = nextResults;
1013 //                        //        nextResults = new HashSet<ulong>();
1014 //                        //    }
1015 //                        //    else
1016 //                        //    {
1017 //                        //        results.IntersectWith(nextResults);
1018 //                        //        nextResults.Clear();
1019 //                        //    }
1020 //                        //}
1021 //                        var collector1 = new AllUsagesCollector1(Links.Unsync, results);
1022 //                        collector1.Collect(Links.Unsync.GetLink(sequence[0]));
1023 //                        var next = new HashSet<ulong>();
1024 //                        for (var i = 1; i < sequence.Count; i++)
1025 //                        {
1026 //                            var collector = new AllUsagesCollector1(Links.Unsync, next);
1027 //                            collector.Collect(Links.Unsync.GetLink(sequence[i]));
```

```
1028    //
1029    //                           results.IntersectWith(next);
1030    //                           next.Clear();
1031    //                       }
1032    //                       var filteredResults = new HashSet<ulong>();
1033    //                       var matcher = new Matcher(this, sequence, filteredResults, null,
     ↪   readAsElements);
1034    //                       matcher.AddAllPartialMatchedToResultsAndReadAsElements(results.OrderBy(x
     ↪   => x)); // OrderBy is a Hack
1035    //                       return filteredResults;
1036    //                   }
1037    //                   return new HashSet<ulong>();
1038    //               });
1039    //           }
1040    //
1041    //           // Does not work
1042    //           //public HashSet<ulong> GetAllPartiallyMatchingSequences5(HashSet<ulong>
     ↪   readAsElements, params ulong[] sequence)
1043    //           //{
1044    //           //    var visited = new HashSet<ulong>();
1045    //           //    var results = new HashSet<ulong>();
1046    //           //    var matcher = new Matcher(this, sequence, visited, x => { results.Add(x);
     ↪   return true; }, readAsElements);
1047    //           //    var last = sequence.Length - 1;
1048    //           //    for (var i = 0; i < last; i++)
1049    //           //    {
1050    //           //        PartialStepRight(matcher.PartialMatch, sequence[i], sequence[i + 1]);
1051    //           //    }
1052    //           //    return results;
1053    //           //}
1054    //
1055    //           /// <summary>
1056    //           /// <para>
1057    //           /// Gets the all partially matching sequences using the specified sequence.
1058    //           /// </para>
1059    //           /// <para></para>
1060    //           /// </summary>
1061    //           /// <param name="sequence">
1062    //           /// <para>The sequence.</para>
1063    //           /// <para></para>
1064    //           /// </param>
1065    //           /// <returns>
1066    //           /// <para>A list of ulong</para>
1067    //           /// <para></para>
1068    //           /// </returns>
1069    //           [MethodImpl(MethodImplOptions.AggressiveInlining)]
1070    //           public List<ulong> GetAllPartiallyMatchingSequences(params ulong[] sequence)
1071    //           {
1072    //               return _sync.DoRead(() =>
1073    //               {
1074    //                   if (sequence.Length > 0)
1075    //                   {
1076    //                       Links.EnsureLinkExists(sequence);
1077    //                       //var firstElement = sequence[0];
1078    //                       //if (sequence.Length == 1)
1079    //                       //{
1080    //                       //    //results.Add(firstElement);
1081    //                       //    return results;
1082    //                       //}
1083    //                       //if (sequence.Length == 2)
1084    //                       //{
1085    //                       //    //var doublet = _links.SearchCore(firstElement, sequence[1]);
1086    //                       //    //if (doublet != Doublets.Links.Null)
1087    //                       //    //    results.Add(doublet);
1088    //                       //    return results;
1089    //                       //}
1090    //                       //var lastElement = sequence[sequence.Length - 1];
1091    //                       //Func<ulong, bool> handler = x =>
1092    //                       //{
1093    //                       //    if (StartsWith(x, firstElement) && EndsWith(x, lastElement))
     ↪   results.Add(x);
1094    //                       //    return true;
1095    //                       //};
1096    //                       //if (sequence.Length >= 2)
1097    //                       //    StepRight(handler, sequence[0], sequence[1]);
1098    //                       //var last = sequence.Length - 2;
1099    //                       //for (var i = 1; i < last; i++)
```

```
1100  //                        //    PartialStepRight(handler, sequence[i], sequence[i + 1]);
1101  //                        //if (sequence.Length >= 3)
1102  //                        //    StepLeft(handler, sequence[sequence.Length - 2],
      ↪ sequence[sequence.Length - 1]);
1103  //                        //////if (sequence.Length == 1)
1104  //                        //////{
1105  //                        //////    throw new NotImplementedException(); // all sequences,
      ↪ containing this element?
1106  //                        //////}
1107  //                        //////if (sequence.Length == 2)
1108  //                        //////{
1109  //                        //////    var results = new List<ulong>();
1110  //                        //////    PartialStepRight(results.Add, sequence[0], sequence[1]);
1111  //                        //////    return results;
1112  //                        //////}
1113  //                        //////var matches = new List<List<ulong>>();
1114  //                        //////var last = sequence.Length - 1;
1115  //                        //////for (var i = 0; i < last; i++)
1116  //                        //////{
1117  //                        //////    var results = new List<ulong>();
1118  //                        //////    //StepRight(results.Add, sequence[i], sequence[i + 1]);
1119  //                        //////    PartialStepRight(results.Add, sequence[i], sequence[i + 1]);
1120  //                        //////    if (results.Count > 0)
1121  //                        //////        matches.Add(results);
1122  //                        //////    else
1123  //                        //////        return results;
1124  //                        //////    if (matches.Count == 2)
1125  //                        //////    {
1126  //                        //////        var merged = new List<ulong>();
1127  //                        //////        for (var j = 0; j < matches[0].Count; j++)
1128  //                        //////            for (var k = 0; k < matches[1].Count; k++)
1129  //                        //////                CloseInnerConnections(merged.Add, matches[0][j],
      ↪ matches[1][k]);
1130  //                        //////        if (merged.Count > 0)
1131  //                        //////            matches = new List<List<ulong>> { merged };
1132  //                        //////        else
1133  //                        //////            return new List<ulong>();
1134  //                        //////    }
1135  //                        //////}
1136  //                        //////if (matches.Count > 0)
1137  //                        //////{
1138  //                        //////    var usages = new HashSet<ulong>();
1139  //                        //////    for (int i = 0; i < sequence.Length; i++)
1140  //                        //////    {
1141  //                        //////        AllUsagesCore(sequence[i], usages);
1142  //                        //////    }
1143  //                        //////    //for (int i = 0; i < matches[0].Count; i++)
1144  //                        //////    //    AllUsagesCore(matches[0][i], usages);
1145  //                        //////    //usages.UnionWith(matches[0]);
1146  //                        //////    return usages.ToList();
1147  //                        //////}
1148  //                        var firstLinkUsages = new HashSet<ulong>();
1149  //                        AllUsagesCore(sequence[0], firstLinkUsages);
1150  //                        firstLinkUsages.Add(sequence[0]);
1151  //                        //var previousMatchings = firstLinkUsages.ToList(); //new List<ulong>() {
      ↪ sequence[0] }; // or all sequences, containing this element?
1152  //                        //return GetAllPartiallyMatchingSequencesCore(sequence, firstLinkUsages,
      ↪ 1).ToList();
1153  //                        var results = new HashSet<ulong>();
1154  //                        foreach (var match in GetAllPartiallyMatchingSequencesCore(sequence,
      ↪ firstLinkUsages, 1))
1155  //                        {
1156  //                            AllUsagesCore(match, results);
1157  //                        }
1158  //                        return results.ToList();
1159  //                    }
1160  //                return new List<ulong>();
1161  //            });
1162  //        }
1163  //
1164  //        /// <remarks>
1165  //        /// TODO: Может потребоваться ограничение на уровень глубины рекурсии
1166  //        /// </remarks>
1167  //        [MethodImpl(MethodImplOptions.AggressiveInlining)]
1168  //        public HashSet<ulong> AllUsages(ulong link)
1169  //        {
1170  //            return _sync.DoRead(() =>
```

```csharp
//                    {
//                        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
//                [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.DoRead(() =>
//                    {
//                        var visits = new HashSet<ulong>();
//                        var usages = new HashSet<ulong>();
//                        AllBottomUsagesCore(link, visits, usages);
//                        return usages;
//                    });
//                }
//                [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>
```

```
1246  //          /// <para></para>
1247  //          /// </summary>
1248  //          /// <param name="symbol">
1249  //          /// <para>The symbol.</para>
1250  //          /// <para></para>
1251  //          /// </param>
1252  //          /// <returns>
1253  //          /// <para>The ulong</para>
1254  //          /// <para></para>
1255  //          /// </returns>
1256  //          [MethodImpl(MethodImplOptions.AggressiveInlining)]
1257  //          public ulong CalculateTotalSymbolFrequencyCore(ulong symbol)
1258  //          {
1259  //              if (Options.UseSequenceMarker)
1260  //              {
1261  //                  var counter = new
      ↪ TotalMarkedSequenceSymbolFrequencyOneOffCounter<ulong>(Links, Options.MarkedSequenceMatcher,
      ↪ symbol);
1262  //                  return counter.Count();
1263  //              }
1264  //              else
1265  //              {
1266  //                  var counter = new TotalSequenceSymbolFrequencyOneOffCounter<ulong>(Links,
      ↪ symbol);
1267  //                  return counter.Count();
1268  //              }
1269  //          }
1270  //          [MethodImpl(MethodImplOptions.AggressiveInlining)]
1271  //          private bool AllUsagesCore1(ulong link, HashSet<ulong> usages, Func<IList<LinkIndex>,
      ↪ LinkIndex> outerHandler)
1272  //          {
1273  //              bool handler(ulong doublet)
1274  //              {
1275  //                  if (usages.Add(doublet))
1276  //                  {
1277  //                      if (outerHandler(new LinkAddress<LinkIndex>(doublet)) !=
      ↪ Constants.Continue)
1278  //                      {
1279  //                          return false;
1280  //                      }
1281  //                      if (!AllUsagesCore1(doublet, usages, outerHandler))
1282  //                      {
1283  //                          return false;
1284  //                      }
1285  //                  }
1286  //                  return true;
1287  //              }
1288  //              return Links.Unsync.Each(link, Constants.Any, handler)
1289  //                  && Links.Unsync.Each(Constants.Any, link, handler);
1290  //          }
1291  //
1292  //          /// <summary>
1293  //          /// <para>
1294  //          /// Calculates the all usages using the specified totals.
1295  //          /// </para>
1296  //          /// <para></para>
1297  //          /// </summary>
1298  //          /// <param name="totals">
1299  //          /// <para>The totals.</para>
1300  //          /// <para></para>
1301  //          /// </param>
1302  //          [MethodImpl(MethodImplOptions.AggressiveInlining)]
1303  //          public void CalculateAllUsages(ulong[] totals)
1304  //          {
1305  //              var calculator = new AllUsagesCalculator(Links, totals);
1306  //              calculator.Calculate();
1307  //          }
1308  //
1309  //          /// <summary>
1310  //          /// <para>
1311  //          /// Calculates the all usages 2 using the specified totals.
1312  //          /// </para>
1313  //          /// <para></para>
1314  //          /// </summary>
1315  //          /// <param name="totals">
1316  //          /// <para>The totals.</para>
1317  //          /// <para></para>
```

```csharp
1318  //           /// </param>
1319  //           [MethodImpl(MethodImplOptions.AggressiveInlining)]
1320  //           public void CalculateAllUsages2(ulong[] totals)
1321  //           {
1322  //               var calculator = new AllUsagesCalculator2(Links, totals);
1323  //               calculator.Calculate();
1324  //           }
1325  //           private class AllUsagesCalculator
1326  //           {
1327  //               private readonly SynchronizedLinks<ulong> _links;
1328  //               private readonly ulong[] _totals;
1329  //
1330  //               /// <summary>
1331  //               /// <para>
1332  //               /// Initializes a new <see cref="AllUsagesCalculator"/> instance.
1333  //               /// </para>
1334  //               /// <para></para>
1335  //               /// </summary>
1336  //               /// <param name="links">
1337  //               /// <para>A links.</para>
1338  //               /// <para></para>
1339  //               /// </param>
1340  //               /// <param name="totals">
1341  //               /// <para>A totals.</para>
1342  //               /// <para></para>
1343  //               /// </param>
1344  //               [MethodImpl(MethodImplOptions.AggressiveInlining)]
1345  //               public AllUsagesCalculator(SynchronizedLinks<ulong> links, ulong[] totals)
1346  //               {
1347  //                   _links = links;
1348  //                   _totals = totals;
1349  //               }
1350  //
1351  //               /// <summary>
1352  //               /// <para>
1353  //               /// Calculates this instance.
1354  //               /// </para>
1355  //               /// <para></para>
1356  //               /// </summary>
1357  //               [MethodImpl(MethodImplOptions.AggressiveInlining)]
1358  //               public void Calculate() => _links.Each(_links.Constants.Any,
      ↪  _links.Constants.Any, CalculateCore);
1359  //               [MethodImpl(MethodImplOptions.AggressiveInlining)]
1360  //               private bool CalculateCore(ulong link)
1361  //               {
1362  //                   if (_totals[link] == 0)
1363  //                   {
1364  //                       var total = 1UL;
1365  //                       _totals[link] = total;
1366  //                       var visitedChildren = new HashSet<ulong>();
1367  //                       bool linkCalculator(ulong child)
1368  //                       {
1369  //                           if (link != child && visitedChildren.Add(child))
1370  //                           {
1371  //                               total += _totals[child] == 0 ? 1 : _totals[child];
1372  //                           }
1373  //                           return true;
1374  //                       }
1375  //                       _links.Unsync.Each(link, _links.Constants.Any, linkCalculator);
1376  //                       _links.Unsync.Each(_links.Constants.Any, link, linkCalculator);
1377  //                       _totals[link] = total;
1378  //                   }
1379  //                   return true;
1380  //               }
1381  //           }
1382  //           private class AllUsagesCalculator2
1383  //           {
1384  //               private readonly SynchronizedLinks<ulong> _links;
1385  //               private readonly ulong[] _totals;
1386  //
1387  //               /// <summary>
1388  //               /// <para>
1389  //               /// Initializes a new <see cref="AllUsagesCalculator2"/> instance.
1390  //               /// </para>
1391  //               /// <para></para>
1392  //               /// </summary>
1393  //               /// <param name="links">
```

```
1394  //                /// <para>A links.</para>
1395  //                /// <para></para>
1396  //                /// </param>
1397  //                /// <param name="totals">
1398  //                /// <para>A totals.</para>
1399  //                /// <para></para>
1400  //                /// </param>
1401  //                [MethodImpl(MethodImplOptions.AggressiveInlining)]
1402  //                public AllUsagesCalculator2(SynchronizedLinks<ulong> links, ulong[] totals)
1403  //                {
1404  //                    _links = links;
1405  //                    _totals = totals;
1406  //                }
1407  //
1408  //                /// <summary>
1409  //                /// <para>
1410  //                /// Calculates this instance.
1411  //                /// </para>
1412  //                /// <para></para>
1413  //                /// </summary>
1414  //                [MethodImpl(MethodImplOptions.AggressiveInlining)]
1415  //                public void Calculate() => _links.Each(_links.Constants.Any,
      ↪  _links.Constants.Any, CalculateCore);
1416  //                [MethodImpl(MethodImplOptions.AggressiveInlining)]
1417  //                private bool IsElement(ulong link)
1418  //                {
1419  //                    //_linksInSequence.Contains(link) ||
1420  //                    return _links.Unsync.GetTarget(link) == link || _links.Unsync.GetSource(link)
      ↪  == link;
1421  //                }
1422  //                [MethodImpl(MethodImplOptions.AggressiveInlining)]
1423  //                private bool CalculateCore(ulong link)
1424  //                {
1425  //                    // TODO: Проработать защиту от зацикливания
1426  //                    // Основано на SequenceWalker.WalkLeft
1427  //                    Func<ulong, ulong> getSource = _links.Unsync.GetSource;
1428  //                    Func<ulong, ulong> getTarget = _links.Unsync.GetTarget;
1429  //                    Func<ulong, bool> isElement = IsElement;
1430  //                    void visitLeaf(ulong parent)
1431  //                    {
1432  //                        if (link != parent)
1433  //                        {
1434  //                            _totals[parent]++;
1435  //                        }
1436  //                    }
1437  //                    void visitNode(ulong parent)
1438  //                    {
1439  //                        if (link != parent)
1440  //                        {
1441  //                            _totals[parent]++;
1442  //                        }
1443  //                    }
1444  //                    var stack = new Stack();
1445  //                    var element = link;
1446  //                    if (isElement(element))
1447  //                    {
1448  //                        visitLeaf(element);
1449  //                    }
1450  //                    else
1451  //                    {
1452  //                        while (true)
1453  //                        {
1454  //                            if (isElement(element))
1455  //                            {
1456  //                                if (stack.Count == 0)
1457  //                                {
1458  //                                    break;
1459  //                                }
1460  //                                element = stack.Pop();
1461  //                                var source = getSource(element);
1462  //                                var target = getTarget(element);
1463  //                                // Обработка элемента
1464  //                                if (isElement(target))
1465  //                                {
1466  //                                    visitLeaf(target);
1467  //                                }
1468  //                                if (isElement(source))
```

```
 1469  //                              {
 1470  //                                      visitLeaf(source);
 1471  //                              }
 1472  //                              element = source;
 1473  //                          }
 1474  //                          else
 1475  //                          {
 1476  //                              stack.Push(element);
 1477  //                              visitNode(element);
 1478  //                              element = getTarget(element);
 1479  //                          }
 1480  //                      }
 1481  //                  }
 1482  //                  _totals[link]++;
 1483  //                  return true;
 1484  //              }
 1485  //          }
 1486  //          private class AllUsagesCollector
 1487  //          {
 1488  //              private readonly ILinks<ulong> _links;
 1489  //              private readonly HashSet<ulong> _usages;
 1490  //
 1491  //              /// <summary>
 1492  //              /// <para>
 1493  //              /// Initializes a new <see cref="AllUsagesCollector"/> instance.
 1494  //              /// </para>
 1495  //              /// <para></para>
 1496  //              /// </summary>
 1497  //              /// <param name="links">
 1498  //              /// <para>A links.</para>
 1499  //              /// <para></para>
 1500  //              /// </param>
 1501  //              /// <param name="usages">
 1502  //              /// <para>A usages.</para>
 1503  //              /// <para></para>
 1504  //              /// </param>
 1505  //              [MethodImpl(MethodImplOptions.AggressiveInlining)]
 1506  //              public AllUsagesCollector(ILinks<ulong> links, HashSet<ulong> usages)
 1507  //              {
 1508  //                  _links = links;
 1509  //                  _usages = usages;
 1510  //              }
 1511  //
 1512  //              /// <summary>
 1513  //              /// <para>
 1514  //              /// Determines whether this instance collect.
 1515  //              /// </para>
 1516  //              /// <para></para>
 1517  //              /// </summary>
 1518  //              /// <param name="link">
 1519  //              /// <para>The link.</para>
 1520  //              /// <para></para>
 1521  //              /// </param>
 1522  //              /// <returns>
 1523  //              /// <para>The bool</para>
 1524  //              /// <para></para>
 1525  //              /// </returns>
 1526  //              [MethodImpl(MethodImplOptions.AggressiveInlining)]
 1527  //              public bool Collect(ulong link)
 1528  //              {
 1529  //                  if (_usages.Add(link))
 1530  //                  {
 1531  //                      _links.Each(link, _links.Constants.Any, Collect);
 1532  //                      _links.Each(_links.Constants.Any, link, Collect);
 1533  //                  }
 1534  //                  return true;
 1535  //              }
 1536  //          }
 1537  //          private class AllUsagesCollector1
 1538  //          {
 1539  //              private readonly ILinks<ulong> _links;
 1540  //              private readonly HashSet<ulong> _usages;
 1541  //              private readonly ulong _continue;
 1542  //
 1543  //              /// <summary>
 1544  //              /// <para>
 1545  //              /// Initializes a new <see cref="AllUsagesCollector1"/> instance.
 1546  //              /// </para>
```

```
1547  //                    /// <para></para>
1548  //                    /// </summary>
1549  //                    /// <param name="links">
1550  //                    /// <para>A links.</para>
1551  //                    /// <para></para>
1552  //                    /// </param>
1553  //                    /// <param name="usages">
1554  //                    /// <para>A usages.</para>
1555  //                    /// <para></para>
1556  //                    /// </param>
1557  //                    [MethodImpl(MethodImplOptions.AggressiveInlining)]
1558  //                    public AllUsagesCollector1(ILinks<ulong> links, HashSet<ulong> usages)
1559  //                    {
1560  //                        _links = links;
1561  //                        _usages = usages;
1562  //                        _continue = _links.Constants.Continue;
1563  //                    }
1564  //
1565  //                    /// <summary>
1566  //                    /// <para>
1567  //                    /// Collects the link.
1568  //                    /// </para>
1569  //                    /// <para></para>
1570  //                    /// </summary>
1571  //                    /// <param name="link">
1572  //                    /// <para>The link.</para>
1573  //                    /// <para></para>
1574  //                    /// </param>
1575  //                    /// <returns>
1576  //                    /// <para>The continue.</para>
1577  //                    /// <para></para>
1578  //                    /// </returns>
1579  //                    [MethodImpl(MethodImplOptions.AggressiveInlining)]
1580  //                    public ulong Collect(IList<ulong> link)
1581  //                    {
1582  //                        var linkIndex = _links.GetIndex(link);
1583  //                        if (_usages.Add(linkIndex))
1584  //                        {
1585  //                            _links.Each(Collect, _links.Constants.Any, linkIndex);
1586  //                        }
1587  //                        return _continue;
1588  //                    }
1589  //                }
1590  //            private class AllUsagesCollector2
1591  //            {
1592  //                private readonly ILinks<ulong> _links;
1593  //                private readonly BitString _usages;
1594  //
1595  //                /// <summary>
1596  //                /// <para>
1597  //                /// Initializes a new <see cref="AllUsagesCollector2"/> instance.
1598  //                /// </para>
1599  //                /// <para></para>
1600  //                /// </summary>
1601  //                /// <param name="links">
1602  //                /// <para>A links.</para>
1603  //                /// <para></para>
1604  //                /// </param>
1605  //                /// <param name="usages">
1606  //                /// <para>A usages.</para>
1607  //                /// <para></para>
1608  //                /// </param>
1609  //                [MethodImpl(MethodImplOptions.AggressiveInlining)]
1610  //                public AllUsagesCollector2(ILinks<ulong> links, BitString usages)
1611  //                {
1612  //                    _links = links;
1613  //                    _usages = usages;
1614  //                }
1615  //
1616  //                /// <summary>
1617  //                /// <para>
1618  //                /// Determines whether this instance collect.
1619  //                /// </para>
1620  //                /// <para></para>
1621  //                /// </summary>
1622  //                /// <param name="link">
1623  //                /// <para>The link.</para>
1624  //                /// <para></para>
```

```
1625  //              /// </param>
1626  //              /// <returns>
1627  //              /// <para>The bool</para>
1628  //              /// <para></para>
1629  //              /// </returns>
1630  //              [MethodImpl(MethodImplOptions.AggressiveInlining)]
1631  //              public bool Collect(ulong link)
1632  //              {
1633  //                  if (_usages.Add((long)link))
1634  //                  {
1635  //                      _links.Each(link, _links.Constants.Any, Collect);
1636  //                      _links.Each(_links.Constants.Any, link, Collect);
1637  //                  }
1638  //                  return true;
1639  //              }
1640  //          }
1641  //          private class AllUsagesIntersectingCollector
1642  //          {
1643  //              private readonly SynchronizedLinks<ulong> _links;
1644  //              private readonly HashSet<ulong> _intersectWith;
1645  //              private readonly HashSet<ulong> _usages;
1646  //              private readonly HashSet<ulong> _enter;
1647  //
1648  //              /// <summary>
1649  //              /// <para>
1650  //              /// Initializes a new <see cref="AllUsagesIntersectingCollector"/> instance.
1651  //              /// </para>
1652  //              /// <para></para>
1653  //              /// </summary>
1654  //              /// <param name="links">
1655  //              /// <para>A links.</para>
1656  //              /// <para></para>
1657  //              /// </param>
1658  //              /// <param name="intersectWith">
1659  //              /// <para>A intersect with.</para>
1660  //              /// <para></para>
1661  //              /// </param>
1662  //              /// <param name="usages">
1663  //              /// <para>A usages.</para>
1664  //              /// <para></para>
1665  //              /// </param>
1666  //              [MethodImpl(MethodImplOptions.AggressiveInlining)]
1667  //              public AllUsagesIntersectingCollector(SynchronizedLinks<ulong> links,
      ↪  HashSet<ulong> intersectWith, HashSet<ulong> usages)
1668  //              {
1669  //                  _links = links;
1670  //                  _intersectWith = intersectWith;
1671  //                  _usages = usages;
1672  //                  _enter = new HashSet<ulong>(); // защита от зацикливания
1673  //              }
1674  //
1675  //              /// <summary>
1676  //              /// <para>
1677  //              /// Determines whether this instance collect.
1678  //              /// </para>
1679  //              /// <para></para>
1680  //              /// </summary>
1681  //              /// <param name="link">
1682  //              /// <para>The link.</para>
1683  //              /// <para></para>
1684  //              /// </param>
1685  //              /// <returns>
1686  //              /// <para>The bool</para>
1687  //              /// <para></para>
1688  //              /// </returns>
1689  //              [MethodImpl(MethodImplOptions.AggressiveInlining)]
1690  //              public bool Collect(ulong link)
1691  //              {
1692  //                  if (_enter.Add(link))
1693  //                  {
1694  //                      if (_intersectWith.Contains(link))
1695  //                      {
1696  //                          _usages.Add(link);
1697  //                      }
1698  //                      _links.Unsync.Each(link, _links.Constants.Any, Collect);
1699  //                      _links.Unsync.Each(_links.Constants.Any, link, Collect);
1700  //                  }
```

```
1701  //                       return true;
1702  //                   }
1703  //               }
1704  //           [MethodImpl(MethodImplOptions.AggressiveInlining)]
1705  //           private void CloseInnerConnections(Action<IList<LinkIndex>> handler, ulong left,
      ↪   ulong right)
1706  //           {
1707  //               TryStepLeftUp(handler, left, right);
1708  //               TryStepRightUp(handler, right, left);
1709  //           }
1710  //           [MethodImpl(MethodImplOptions.AggressiveInlining)]
1711  //           private void AllCloseConnections(Action<IList<LinkIndex>> handler, ulong left, ulong
      ↪   right)
1712  //           {
1713  //               // Direct
1714  //               if (left == right)
1715  //               {
1716  //                   handler(new LinkAddress<LinkIndex>(left));
1717  //               }
1718  //               var doublet = Links.Unsync.SearchOrDefault(left, right);
1719  //               if (doublet != Constants.Null)
1720  //               {
1721  //                   handler(new LinkAddress<LinkIndex>(doublet));
1722  //               }
1723  //               // Inner
1724  //               CloseInnerConnections(handler, left, right);
1725  //               // Outer
1726  //               StepLeft(handler, left, right);
1727  //               StepRight(handler, left, right);
1728  //               PartialStepRight(handler, left, right);
1729  //               PartialStepLeft(handler, left, right);
1730  //           }
1731  //           [MethodImpl(MethodImplOptions.AggressiveInlining)]
1732  //           private HashSet<ulong> GetAllPartiallyMatchingSequencesCore(ulong[] sequence,
      ↪   HashSet<ulong> previousMatchings, long startAt)
1733  //           {
1734  //               if (startAt >= sequence.Length) // ?
1735  //               {
1736  //                   return previousMatchings;
1737  //               }
1738  //               var secondLinkUsages = new HashSet<ulong>();
1739  //               AllUsagesCore(sequence[startAt], secondLinkUsages);
1740  //               secondLinkUsages.Add(sequence[startAt]);
1741  //               var matchings = new HashSet<ulong>();
1742  //               var filler = new SetFiller<LinkIndex, LinkIndex>(matchings, Constants.Continue);
1743  //               //for (var i = 0; i < previousMatchings.Count; i++)
1744  //               foreach (var secondLinkUsage in secondLinkUsages)
1745  //               {
1746  //                   foreach (var previousMatching in previousMatchings)
1747  //                   {
1748  //                       //AllCloseConnections(matchings.AddAndReturnVoid, previousMatching,
      ↪   secondLinkUsage);
1749  //                       StepRight(filler.AddFirstAndReturnConstant, previousMatching,
      ↪   secondLinkUsage);
1750  //                       TryStepRightUp(filler.AddFirstAndReturnConstant, secondLinkUsage,
      ↪   previousMatching);
1751  //                       //PartialStepRight(matchings.AddAndReturnVoid, secondLinkUsage,
      ↪   sequence[startAt]); // почему-то эта ошибочная запись приводит к желаемым результам.
1752  //                       PartialStepRight(filler.AddFirstAndReturnConstant, previousMatching,
      ↪   secondLinkUsage);
1753  //                   }
1754  //               }
1755  //               if (matchings.Count == 0)
1756  //               {
1757  //                   return matchings;
1758  //               }
1759  //               return GetAllPartiallyMatchingSequencesCore(sequence, matchings, startAt + 1); //
      ↪   ??
1760  //           }
1761  //           [MethodImpl(MethodImplOptions.AggressiveInlining)]
1762  //           private static void EnsureEachLinkIsAnyOrZeroOrManyOrExists(SynchronizedLinks<ulong>
      ↪   links, params ulong[] sequence)
1763  //           {
1764  //               if (sequence == null)
1765  //               {
1766  //                   return;
1767  //               }
```

```
1768  //                    for (var i = 0; i < sequence.Length; i++)
1769  //                    {
1770  //                        if (sequence[i] != links.Constants.Any && sequence[i] != ZeroOrMany &&
      ↪  !links.Exists(sequence[i]))
1771  //                        {
1772  //                            throw new ArgumentLinkDoesNotExistsException<ulong>(sequence[i],
      ↪  $"patternSequence[{i}]");
1773  //                        }
1774  //                    }
1775  //            }
1776  //
1777  //            // Pattern Matching -> Key To Triggers
1778  //            /// <summary>
1779  //            /// <para>
1780  //            /// Matches the pattern using the specified pattern sequence.
1781  //            /// </para>
1782  //            /// <para></para>
1783  //            /// </summary>
1784  //            /// <param name="patternSequence">
1785  //            /// <para>The pattern sequence.</para>
1786  //            /// <para></para>
1787  //            /// </param>
1788  //            /// <returns>
1789  //            /// <para>A hash set of ulong</para>
1790  //            /// <para></para>
1791  //            /// </returns>
1792  //            [MethodImpl(MethodImplOptions.AggressiveInlining)]
1793  //            public HashSet<ulong> MatchPattern(params ulong[] patternSequence)
1794  //            {
1795  //                return _sync.DoRead(() =>
1796  //                {
1797  //                    patternSequence = Simplify(patternSequence);
1798  //                    if (patternSequence.Length > 0)
1799  //                    {
1800  //                        EnsureEachLinkIsAnyOrZeroOrManyOrExists(Links, patternSequence);
1801  //                        var uniqueSequenceElements = new HashSet<ulong>();
1802  //                        for (var i = 0; i < patternSequence.Length; i++)
1803  //                        {
1804  //                            if (patternSequence[i] != Constants.Any && patternSequence[i] !=
      ↪  ZeroOrMany)
1805  //                            {
1806  //                                uniqueSequenceElements.Add(patternSequence[i]);
1807  //                            }
1808  //                        }
1809  //                        var results = new HashSet<ulong>();
1810  //                        foreach (var uniqueSequenceElement in uniqueSequenceElements)
1811  //                        {
1812  //                            AllUsagesCore(uniqueSequenceElement, results);
1813  //                        }
1814  //                        var filteredResults = new HashSet<ulong>();
1815  //                        var matcher = new PatternMatcher(this, patternSequence, filteredResults);
1816  //                        matcher.AddAllPatternMatchedToResults(results);
1817  //                        return filteredResults;
1818  //                    }
1819  //                    return new HashSet<ulong>();
1820  //                });
1821  //            }
1822  //
1823  //            // Найти все возможные связи между указанным списком связей.
1824  //            // Находит связи между всеми указанными связями в любом порядке.
1825  //            // TODO: решить что делать с повторами (когда одни и те же элементы встречаются
      ↪  несколько раз в последовательности)
1826  //            /// <summary>
1827  //            /// <para>
1828  //            /// Gets the all connections using the specified links to connect.
1829  //            /// </para>
1830  //            /// <para></para>
1831  //            /// </summary>
1832  //            /// <param name="linksToConnect">
1833  //            /// <para>The links to connect.</para>
1834  //            /// <para></para>
1835  //            /// </param>
1836  //            /// <returns>
1837  //            /// <para>A hash set of ulong</para>
1838  //            /// <para></para>
1839  //            /// </returns>
1840  //            [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```csharp
//         public HashSet<ulong> GetAllConnections(params ulong[] linksToConnect)
//         {
//             return _sync.DoRead(() =>
//             {
//                 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.DoRead(() =>
//             {
//                 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.DoRead(() =>
//             {
//                 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.DoRead(() =>
//                {
//                    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();
//                });
//            }
//            [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
//          }
//
```

```csharp
//          #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;
//          }
//          [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];
```

```
2143  //                      var elements = GetLeftElements(rightLink, nextRightLink);
2144  //                      if (leftBound <= rightBound)
2145  //                      {
2146  //                          for (var i = elements.Length - 1; i >= 0; i--)
2147  //                          {
2148  //                              var element = elements[i];
2149  //                              if (element != 0)
2150  //                              {
2151  //                                  CollectMatchingSequences(leftLink, leftBound, middleLinks,
       ↪  elements[i], rightBound - 1, ref results);
2152  //                              }
2153  //                          }
2154  //                      }
2155  //                      else
2156  //                      {
2157  //                          for (var i = elements.Length - 1; i >= 0; i--)
2158  //                          {
2159  //                              var element = elements[i];
2160  //                              if (element != 0)
2161  //                              {
2162  //                                  results.Add(element);
2163  //                              }
2164  //                          }
2165  //                      }
2166  //                  }
2167  //              }
2168  //
2169  //          /// <summary>
2170  //          /// <para>
2171  //          /// Gets the right elements using the specified start link.
2172  //          /// </para>
2173  //          /// <para></para>
2174  //          /// </summary>
2175  //          /// <param name="startLink">
2176  //          /// <para>The start link.</para>
2177  //          /// <para></para>
2178  //          /// </param>
2179  //          /// <param name="rightLink">
2180  //          /// <para>The right link.</para>
2181  //          /// <para></para>
2182  //          /// </param>
2183  //          /// <returns>
2184  //          /// <para>The result.</para>
2185  //          /// <para></para>
2186  //          /// </returns>
2187  //          [MethodImpl(MethodImplOptions.AggressiveInlining)]
2188  //          public ulong[] GetRightElements(ulong startLink, ulong rightLink)
2189  //          {
2190  //              var result = new ulong[5];
2191  //              TryStepRight(startLink, rightLink, result, 0);
2192  //              Links.Each(Constants.Any, startLink, couple =>
2193  //              {
2194  //                  if (couple != startLink)
2195  //                  {
2196  //                      if (TryStepRight(couple, rightLink, result, 2))
2197  //                      {
2198  //                          return false;
2199  //                      }
2200  //                  }
2201  //                  return true;
2202  //              });
2203  //              if (Links.GetTarget(Links.GetTarget(startLink)) == rightLink)
2204  //              {
2205  //                  result[4] = startLink;
2206  //              }
2207  //              return result;
2208  //          }
2209  //
2210  //          /// <summary>
2211  //          /// <para>
2212  //          /// Determines whether this instance try step right.
2213  //          /// </para>
2214  //          /// <para></para>
2215  //          /// </summary>
2216  //          /// <param name="startLink">
2217  //          /// <para>The start link.</para>
2218  //          /// <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))
//                       {
```

```
2296  //                            return false;
2297  //                        }
2298  //                    }
2299  //                    return true;
2300  //                });
2301  //                if (Links.GetSource(Links.GetSource(leftLink)) == startLink)
2302  //                {
2303  //                    result[4] = leftLink;
2304  //                }
2305  //                return result;
2306  //            }
2307  //
2308  //            /// <summary>
2309  //            /// <para>
2310  //            /// Determines whether this instance try step left.
2311  //            /// </para>
2312  //            /// <para></para>
2313  //            /// </summary>
2314  //            /// <param name="startLink">
2315  //            /// <para>The start link.</para>
2316  //            /// <para></para>
2317  //            /// </param>
2318  //            /// <param name="leftLink">
2319  //            /// <para>The left link.</para>
2320  //            /// <para></para>
2321  //            /// </param>
2322  //            /// <param name="result">
2323  //            /// <para>The result.</para>
2324  //            /// <para></para>
2325  //            /// </param>
2326  //            /// <param name="offset">
2327  //            /// <para>The offset.</para>
2328  //            /// <para></para>
2329  //            /// </param>
2330  //            /// <returns>
2331  //            /// <para>The bool</para>
2332  //            /// <para></para>
2333  //            /// </returns>
2334  //            [MethodImpl(MethodImplOptions.AggressiveInlining)]
2335  //            public bool TryStepLeft(ulong startLink, ulong leftLink, ulong[] result, int offset)
2336  //            {
2337  //                var added = 0;
2338  //                Links.Each(Constants.Any, startLink, couple =>
2339  //                {
2340  //                    if (couple != startLink)
2341  //                    {
2342  //                        var coupleSource = Links.GetSource(couple);
2343  //                        if (coupleSource == leftLink)
2344  //                        {
2345  //                            result[offset] = couple;
2346  //                            if (++added == 2)
2347  //                            {
2348  //                                return false;
2349  //                            }
2350  //                        }
2351  //                        else if (Links.GetTarget(coupleSource) == leftLink) //
      ↪  coupleSource.Linker == Net.And &&
2352  //                        {
2353  //                            result[offset + 1] = couple;
2354  //                            if (++added == 2)
2355  //                            {
2356  //                                return false;
2357  //                            }
2358  //                        }
2359  //                    }
2360  //                    return true;
2361  //                });
2362  //                return added > 0;
2363  //            }
2364  //
2365  //            #endregion
2366  //
2367  //            #region Walkers
2368  //
2369  //            /// <summary>
2370  //            /// <para>
2371  //            /// Represents the pattern matcher.
2372  //            /// </para>
```

```csharp
//            /// <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;
//                          }
//                      }
```

```
2524  //                      return _patternPosition == _pattern.Count || (_patternPosition ==
      ↪   _pattern.Count - 1 && _pattern[_patternPosition].Start == 0);
2525  //              }
2526  //              [MethodImpl(MethodImplOptions.AggressiveInlining)]
2527  //              private List<PatternBlock> CreateDetailedPattern()
2528  //              {
2529  //                  var pattern = new List<PatternBlock>();
2530  //                  var patternBlock = new PatternBlock();
2531  //                  for (var i = 0; i < _patternSequence.Length; i++)
2532  //                  {
2533  //                      if (patternBlock.Type == PatternBlockType.Undefined)
2534  //                      {
2535  //                          if (_patternSequence[i] == _sequences.Constants.Any)
2536  //                          {
2537  //                              patternBlock.Type = PatternBlockType.Gap;
2538  //                              patternBlock.Start = 1;
2539  //                              patternBlock.Stop = 1;
2540  //                          }
2541  //                          else if (_patternSequence[i] == ZeroOrMany)
2542  //                          {
2543  //                              patternBlock.Type = PatternBlockType.Gap;
2544  //                              patternBlock.Start = 0;
2545  //                              patternBlock.Stop = long.MaxValue;
2546  //                          }
2547  //                          else
2548  //                          {
2549  //                              patternBlock.Type = PatternBlockType.Elements;
2550  //                              patternBlock.Start = i;
2551  //                              patternBlock.Stop = i;
2552  //                          }
2553  //                      }
2554  //                      else if (patternBlock.Type == PatternBlockType.Elements)
2555  //                      {
2556  //                          if (_patternSequence[i] == _sequences.Constants.Any)
2557  //                          {
2558  //                              pattern.Add(patternBlock);
2559  //                              patternBlock = new PatternBlock
2560  //                              {
2561  //                                  Type = PatternBlockType.Gap,
2562  //                                  Start = 1,
2563  //                                  Stop = 1
2564  //                              };
2565  //                          }
2566  //                          else if (_patternSequence[i] == ZeroOrMany)
2567  //                          {
2568  //                              pattern.Add(patternBlock);
2569  //                              patternBlock = new PatternBlock
2570  //                              {
2571  //                                  Type = PatternBlockType.Gap,
2572  //                                  Start = 0,
2573  //                                  Stop = long.MaxValue
2574  //                              };
2575  //                          }
2576  //                          else
2577  //                          {
2578  //                              patternBlock.Stop = i;
2579  //                          }
2580  //                      }
2581  //                      else // patternBlock.Type == PatternBlockType.Gap
2582  //                      {
2583  //                          if (_patternSequence[i] == _sequences.Constants.Any)
2584  //                          {
2585  //                              patternBlock.Start++;
2586  //                              if (patternBlock.Stop < patternBlock.Start)
2587  //                              {
2588  //                                  patternBlock.Stop = patternBlock.Start;
2589  //                              }
2590  //                          }
2591  //                          else if (_patternSequence[i] == ZeroOrMany)
2592  //                          {
2593  //                              patternBlock.Stop = long.MaxValue;
2594  //                          }
2595  //                          else
2596  //                          {
2597  //                              pattern.Add(patternBlock);
2598  //                              patternBlock = new PatternBlock
2599  //                              {
```

```
//                              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;
//          //}
//          [MethodImpl(MethodImplOptions.AggressiveInlining)]
//          private bool PatternMatchCore(LinkIndex element)
//          {
//              if (_patternPosition >= _pattern.Count)
//              {
//                  _patternPosition = -2;
//                  return false;
//              }
//              var currentPatternBlock = _pattern[_patternPosition];
//              if (currentPatternBlock.Type == PatternBlockType.Gap)
```

```
//                        {
//                            //var currentMatchingBlockLength = (_sequencePosition -
//    _lastMatchedBlockPosition);
//                            if (_sequencePosition < currentPatternBlock.Start)
//                            {
//                                _sequencePosition++;
//                                return true; // Двигаемся дальше
//                            }
//                            // Это последний блок
//                            if (_pattern.Count == _patternPosition + 1)
//                            {
//                                _patternPosition++;
//                                _sequencePosition = 0;
//                                return false; // Полное соответствие
//                            }
//                            else
//                            {
//                                if (_sequencePosition > currentPatternBlock.Stop)
//                                {
//                                    return false; // Соответствие невозможно
//                                }
//                                var nextPatternBlock = _pattern[_patternPosition + 1];
//                                if (_patternSequence[nextPatternBlock.Start] == element)
//                                {
//                                    if (nextPatternBlock.Start < nextPatternBlock.Stop)
//                                    {
//                                        _patternPosition++;
//                                        _sequencePosition = 1;
//                                    }
//                                    else
//                                    {
//                                        _patternPosition += 2;
//                                        _sequencePosition = 0;
//                                    }
//                                }
//                            }
//                        }
//                        else // currentPatternBlock.Type == PatternBlockType.Elements
//                        {
//                            var patternElementPosition = currentPatternBlock.Start +
//    _sequencePosition;
//                            if (_patternSequence[patternElementPosition] != element)
//                            {
//                                return false; // Соответствие невозможно
//                            }
//                            if (patternElementPosition == currentPatternBlock.Stop)
//                            {
//                                _patternPosition++;
//                                _sequencePosition = 0;
//                            }
//                            else
//                            {
//                                _sequencePosition++;
//                            }
//                        }
//                        return true;
//                        //if (_patternSequence[_patternPosition] != element)
//                        //    return false;
//                        //else
//                        //{
//                        //    _sequencePosition++;
//                        //    _patternPosition++;
//                        //    return true;
//                        //}
//                        ////////
//                        //if (_filterPosition == _patternSequence.Length)
//                        //{
//                        //    _filterPosition = -2; // Длиннее чем нужно
//                        //    return false;
//                        //}
//                        //if (element != _patternSequence[_filterPosition])
//                        //{
//                        //    _filterPosition = -1;
//                        //    return false; // Начинается иначе
//                        //}
//                        //_filterPosition++;
//                        //if (_filterPosition == (_patternSequence.Length - 1))
```

```
2751  //                    //      return false;
2752  //                    //if (_filterPosition >= 0)
2753  //                    //{
2754  //                    //      if (element == _patternSequence[_filterPosition + 1])
2755  //                    //          _filterPosition++;
2756  //                    //      else
2757  //                    //          return false;
2758  //                    //}
2759  //                    //if (_filterPosition < 0)
2760  //                    //{
2761  //                    //      if (element == _patternSequence[0])
2762  //                    //          _filterPosition = 0;
2763  //                    //}
2764  //                    }
2765  //
2766  //                    /// <summary>
2767  //                    /// <para>
2768  //                    /// Adds the all pattern matched to results using the specified sequences to
2769  //  ↪  match.
2769  //                    /// </para>
2770  //                    /// <para></para>
2771  //                    /// </summary>
2772  //                    /// <param name="sequencesToMatch">
2773  //                    /// <para>The sequences to match.</para>
2774  //                    /// <para></para>
2775  //                    /// </param>
2776  //                    [MethodImpl(MethodImplOptions.AggressiveInlining)]
2777  //                    public void AddAllPatternMatchedToResults(IEnumerable<ulong> sequencesToMatch)
2778  //                    {
2779  //                        foreach (var sequenceToMatch in sequencesToMatch)
2780  //                        {
2781  //                            if (PatternMatch(sequenceToMatch))
2782  //                            {
2783  //                                _results.Add(sequenceToMatch);
2784  //                            }
2785  //                        }
2786  //                    }
2787  //                }
2788  //
2789  //            #endregion
2790  //        }
2791  // }
```

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

```
1   // using System;
2   // using System.Collections.Generic;
3   // using System.Linq;
4   // using System.Runtime.CompilerServices;
5   // using Platform.Collections;
6   // using Platform.Collections.Lists;
7   // using Platform.Collections.Stacks;
8   // using Platform.Threading.Synchronization;
9   // using Platform.Data.Doublets.Sequences.Walkers;
10  // using Platform.Delegates;
11  // using LinkIndex = System.UInt64;
12  //
13  // #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
14  //
15  // namespace Platform.Data.Doublets.Sequences
16  // {
17  //     /// <summary>
18  //     /// Представляет коллекцию последовательностей связей.
19  //     /// </summary>
20  //     /// <remarks>
21  //     /// Обязательно реализовать атомарность каждого публичного метода.
22  //     ///
23  //     /// TODO:
24  //     ///
25  //     /// !!! Повышение вероятности повторного использования групп (подпоследовательностей),
26  //     /// через естественную группировку по unicode типам, все whitespace вместе, все символы
27  //  ↪  вместе, все числа вместе и т.п.
27  //     /// + использовать ровно сбалансированный вариант, чтобы уменьшать вложенность (глубину
28  //  ↪  графа)
28  //     ///
29  //     /// x*y - найти все связи между, в последовательностях любой формы, если не стоит
30  //  ↪  ограничитель на то, что является последовательностью, а что нет,
30  //     /// то находятся любые структуры связей, которые содержат эти элементы именно в таком
31  //  ↪  порядке.
```

```
31   //          ///
32   //          /// Рост последовательности слева и справа.
33   //          /// Поиск со звёздочкой.
34   //          /// URL, PURL - реестр используемых во вне ссылок на ресурсы,
35   //          /// так же проблема может быть решена при реализации дистанционных триггеров.
36   //          /// Нужны ли уникальные указатели вообще?
37   //          /// Что если обращение к информации будет происходить через содержимое всегда?
38   //          ///
39   //          /// Писать тесты.
40   //          ///
41   //          ///
42   //          /// Можно убрать зависимость от конкретной реализации Links,
43   //          /// на зависимость от абстрактного элемента, который может быть представлен несколькими
     ↪  способами.
44   //          ///
45   //          /// Можно ли как-то сделать один общий интерфейс
46   //          ///
47   //          ///
48   //          /// Блокчейн и/или гит для распределённой записи транзакций.
49   //          ///
50   //          /// </remarks>
51   //      public partial class Sequences : ILinks<LinkIndex> // IList<string>, IList<LinkIndex[]>
     ↪  (после завершения реализации Sequences)
52   //      {
53   //          /// <summary>Возвращает значение LinkIndex, обозначающее любое количество
     ↪  связей.</summary>
54   //          public const LinkIndex ZeroOrMany = LinkIndex.MaxValue;
55   //
56   //          /// <summary>
57   //          /// <para>
58   //          /// Gets the options value.
59   //          /// </para>
60   //          /// <para></para>
61   //          /// </summary>
62   //          public SequencesOptions<LinkIndex> Options { get; }
63   //          /// <summary>
64   //          /// <para>
65   //          /// Gets the links value.
66   //          /// </para>
67   //          /// <para></para>
68   //          /// </summary>
69   //          public SynchronizedLinks<LinkIndex> Links { get; }
70   //          private readonly ISynchronization _sync;
71   //
72   //          /// <summary>
73   //          /// <para>
74   //          /// Gets the constants value.
75   //          /// </para>
76   //          /// <para></para>
77   //          /// </summary>
78   //          public LinksConstants<LinkIndex> Constants { get; }
79   //
80   //          /// <summary>
81   //          /// <para>
82   //          /// Initializes a new <see cref="Sequences"/> instance.
83   //          /// </para>
84   //          /// <para></para>
85   //          /// </summary>
86   //          /// <param name="links">
87   //          /// <para>A links.</para>
88   //          /// <para></para>
89   //          /// </param>
90   //          /// <param name="options">
91   //          /// <para>A options.</para>
92   //          /// <para></para>
93   //          /// </param>
94   //          [MethodImpl(MethodImplOptions.AggressiveInlining)]
95   //          public Sequences(SynchronizedLinks<LinkIndex> links, SequencesOptions<LinkIndex>
     ↪  options)
96   //          {
97   //              Links = links;
98   //              _sync = links.SyncRoot;
99   //              Options = options;
100  //              Options.ValidateOptions();
101  //              Options.InitOptions(Links);
102  //              Constants = links.Constants;
103  //          }
```

```
104  //
105  //          /// <summary>
106  //          /// <para>
107  //          /// Initializes a new <see cref="Sequences"/> instance.
108  //          /// </para>
109  //          /// <para></para>
110  //          /// </summary>
111  //          /// <param name="links">
112  //          /// <para>A links.</para>
113  //          /// <para></para>
114  //          /// </param>
115  //          [MethodImpl(MethodImplOptions.AggressiveInlining)]
116  //          public Sequences(SynchronizedLinks<LinkIndex> links) : this(links, new
     ↪  SequencesOptions<LinkIndex>()) { }
117  //
118  //          /// <summary>
119  //          /// <para>
120  //          /// Determines whether this instance is sequence.
121  //          /// </para>
122  //          /// <para></para>
123  //          /// </summary>
124  //          /// <param name="sequence">
125  //          /// <para>The sequence.</para>
126  //          /// <para></para>
127  //          /// </param>
128  //          /// <returns>
129  //          /// <para>The bool</para>
130  //          /// <para></para>
131  //          /// </returns>
132  //          [MethodImpl(MethodImplOptions.AggressiveInlining)]
133  //          public bool IsSequence(LinkIndex sequence)
134  //          {
135  //              return _sync.DoRead(() =>
136  //              {
137  //                  if (Options.UseSequenceMarker)
138  //                  {
139  //                      return Options.MarkedSequenceMatcher.IsMatched(sequence);
140  //                  }
141  //                  return !Links.Unsync.IsPartialPoint(sequence);
142  //              });
143  //          }
144  //          [MethodImpl(MethodImplOptions.AggressiveInlining)]
145  //          private LinkIndex GetSequenceByElements(LinkIndex sequence)
146  //          {
147  //              if (Options.UseSequenceMarker)
148  //              {
149  //                  return Links.SearchOrDefault(Options.SequenceMarkerLink, sequence);
150  //              }
151  //              return sequence;
152  //          }
153  //          [MethodImpl(MethodImplOptions.AggressiveInlining)]
154  //          private LinkIndex GetSequenceElements(LinkIndex sequence)
155  //          {
156  //              if (Options.UseSequenceMarker)
157  //              {
158  //                  var linkContents = new Link<ulong>(Links.GetLink(sequence));
159  //                  if (linkContents.Source == Options.SequenceMarkerLink)
160  //                  {
161  //                      return linkContents.Target;
162  //                  }
163  //                  if (linkContents.Target == Options.SequenceMarkerLink)
164  //                  {
165  //                      return linkContents.Source;
166  //                  }
167  //              }
168  //              return sequence;
169  //          }
170  //
171  //          #region Count
172  //
173  //          /// <summary>
174  //          /// <para>
175  //          /// Counts the restriction.
176  //          /// </para>
177  //          /// <para></para>
178  //          /// </summary>
179  //          /// <param name="restriction">
```

```
180  //            /// <para>The restriction.</para>
181  //            /// <para></para>
182  //            /// </param>
183  //            /// <exception cref="NotImplementedException">
184  //            /// <para></para>
185  //            /// <para></para>
186  //            /// </exception>
187  //            /// <returns>
188  //            /// <para>The link index</para>
189  //            /// <para></para>
190  //            /// </returns>
191  //            [MethodImpl(MethodImplOptions.AggressiveInlining)]
192  //            public LinkIndex Count(IList<LinkIndex>? restriction)
193  //            {
194  //                if (restriction.IsNullOrEmpty())
195  //                {
196  //                    return Links.Count(Constants.Any, Options.SequenceMarkerLink, Constants.Any);
197  //                }
198  //                if (restriction.Count == 1) // Первая связь это адрес
199  //                {
200  //                    var sequenceIndex = restriction[0];
201  //                    if (sequenceIndex == Constants.Null)
202  //                    {
203  //                        return 0;
204  //                    }
205  //                    if (sequenceIndex == Constants.Any)
206  //                    {
207  //                        return Count(null);
208  //                    }
209  //                    if (Options.UseSequenceMarker)
210  //                    {
211  //                        return Links.Count(Constants.Any, Options.SequenceMarkerLink,
     ↪ sequenceIndex);
212  //                    }
213  //                    return Links.Exists(sequenceIndex) ? 1UL : 0;
214  //                }
215  //                throw new NotImplementedException();
216  //            }
217  //            [MethodImpl(MethodImplOptions.AggressiveInlining)]
218  //            private LinkIndex CountUsages(params LinkIndex[] restriction)
219  //            {
220  //                if (restriction.Length == 0)
221  //                {
222  //                    return 0;
223  //                }
224  //                if (restriction.Length == 1) // Первая связь это адрес
225  //                {
226  //                    if (restriction[0] == Constants.Null)
227  //                    {
228  //                        return 0;
229  //                    }
230  //                    var any = Constants.Any;
231  //                    if (Options.UseSequenceMarker)
232  //                    {
233  //                        var elementsLink = GetSequenceElements(restriction[0]);
234  //                        var sequenceLink = GetSequenceByElements(elementsLink);
235  //                        if (sequenceLink != Constants.Null)
236  //                        {
237  //                            return Links.Count(any, sequenceLink) + Links.Count(any,
     ↪ elementsLink) - 1;
238  //                        }
239  //                        return Links.Count(any, elementsLink);
240  //                    }
241  //                    return Links.Count(any, restriction[0]);
242  //                }
243  //                throw new NotImplementedException();
244  //            }
245  //
246  //            #endregion
247  //
248  //            #region Create
249  //
250  //            /// <summary>
251  //            /// <para>
252  //            /// Creates the restriction.
253  //            /// </para>
254  //            /// <para></para>
```

```csharp
//            /// </summary>
//            /// <param name="restriction">
//            /// <para>The restriction.</para>
//            /// <para></para>
//            /// </param>
//            /// <returns>
//            /// <para>The link index</para>
//            /// <para></para>
//            /// </returns>
//            [MethodImpl(MethodImplOptions.AggressiveInlining)]
//            public LinkIndex Create(IList<LinkIndex>? restriction)
//            {
//                return _sync.DoWrite(() =>
//                {
//                    if (restriction.IsNullOrEmpty())
//                    {
//                        return Constants.Null;
//                    }
//                    Links.EnsureInnerReferenceExists(restriction, nameof(restriction));
//                    return CreateCore(restriction);
//                });
//            }
//            [MethodImpl(MethodImplOptions.AggressiveInlining)]
//            private LinkIndex CreateCore(IList<LinkIndex>? restriction)
//            {
//                LinkIndex[] sequence = restriction.SkipFirst();
//                if (Options.UseIndex)
//                {
//                    Options.Index.Add(sequence);
//                }
//                var sequenceRoot = default(LinkIndex);
//                if (Options.EnforceSingleSequenceVersionOnWriteBasedOnExisting)
//                {
//                    var matches = Each(restriction);
//                    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);
```

```
333  //                    return results;
334  //            }
335  //
336  //            /// <summary>
337  //            /// <para>
338  //            /// Eaches the handler.
339  //            /// </para>
340  //            /// <para></para>
341  //            /// </summary>
342  //            /// <param name="handler">
343  //            /// <para>The handler.</para>
344  //            /// <para></para>
345  //            /// </param>
346  //            /// <param name="restriction">
347  //            /// <para>The restriction.</para>
348  //            /// <para></para>
349  //            /// </param>
350  //            /// <exception cref="NotImplementedException">
351  //            /// <para></para>
352  //            /// <para></para>
353  //            /// </exception>
354  //            /// <returns>
355  //            /// <para>The link index</para>
356  //            /// <para></para>
357  //            /// </returns>
358  //            [MethodImpl(MethodImplOptions.AggressiveInlining)]
359  //            public LinkIndex Each(ReadHandler<LinkIndex> handler, IList<LinkIndex>? restriction)
360  //            {
361  //                return _sync.DoRead(() =>
362  //                {
363  //                    if (restriction.IsNullOrEmpty())
364  //                    {
365  //                        return Constants.Continue;
366  //                    }
367  //                    Links.EnsureInnerReferenceExists(restriction, nameof(restriction));
368  //                    if (restriction.Count == 1)
369  //                    {
370  //                        var link = restriction[0];
371  //                        var any = Constants.Any;
372  //                        if (link == any)
373  //                        {
374  //                            if (Options.UseSequenceMarker)
375  //                            {
376  //                                return Links.Unsync.Each(new Link<LinkIndex>(any,
     ↪ Options.SequenceMarkerLink, any), handler);
377  //                            }
378  //                            else
379  //                            {
380  //                                return Links.Unsync.Each(handler, new Link<LinkIndex>(any, any,
     ↪ any));
381  //                            }
382  //                        }
383  //                        if (Options.UseSequenceMarker)
384  //                        {
385  //                            var sequenceLinkValues = Links.Unsync.GetLink(link);
386  //                            if (sequenceLinkValues[Constants.SourcePart] ==
     ↪ Options.SequenceMarkerLink)
387  //                            {
388  //                                link = sequenceLinkValues[Constants.TargetPart];
389  //                            }
390  //                        }
391  //                        var sequence = Options.Walker.Walk(link).ToArray().ShiftRight();
392  //                        sequence[0] = link;
393  //                        return handler(sequence);
394  //                    }
395  //                    else if (restriction.Count == 2)
396  //                    {
397  //                        throw new NotImplementedException();
398  //                    }
399  //                    else if (restriction.Count == 3)
400  //                    {
401  //                        return Links.Unsync.Each(restriction, handler);
402  //                    }
403  //                    else
404  //                    {
405  //                        var sequence = restriction.SkipFirst();
406  //                        if (Options.UseIndex && !Options.Index.MightContain(sequence))
```

```
407  //                          {
408  //                              return Constants.Break;
409  //                          }
410  //                          return EachCore(sequence, handler);
411  //                      }
412  //                  });
413  //              }
414  //              [MethodImpl(MethodImplOptions.AggressiveInlining)]
415  //              private LinkIndex EachCore(IList<LinkIndex> values, ReadHandler<LinkIndex> handler)
416  //              {
417  //                  var matcher = new Matcher(this, values, new HashSet<LinkIndex>(), handler);
418  //                  // TODO: Find out why matcher.HandleFullMatched executed twice for the same
     ↪  sequence Id.
419  //                  Func<IList<LinkIndex>, LinkIndex> innerHandler = Options.UseSequenceMarker ?
     ↪  (Func<IList<LinkIndex>, LinkIndex>)matcher.HandleFullMatchedSequence :
     ↪  matcher.HandleFullMatched;
420  //                  //if (sequence.Length >= 2)
421  //                  if (StepRight(innerHandler, values[0], values[1]) != Constants.Continue)
422  //                  {
423  //                      return Constants.Break;
424  //                  }
425  //                  var last = values.Count - 2;
426  //                  for (var i = 1; i < last; i++)
427  //                  {
428  //                      if (PartialStepRight(innerHandler, values[i], values[i + 1]) !=
     ↪  Constants.Continue)
429  //                      {
430  //                          return Constants.Break;
431  //                      }
432  //                  }
433  //                  if (values.Count >= 3)
434  //                  {
435  //                      if (StepLeft(innerHandler, values[values.Count - 2], values[values.Count -
     ↪  1]) != Constants.Continue)
436  //                      {
437  //                          return Constants.Break;
438  //                      }
439  //                  }
440  //                  return Constants.Continue;
441  //              }
442  //              [MethodImpl(MethodImplOptions.AggressiveInlining)]
443  //              private LinkIndex PartialStepRight(Func<IList<LinkIndex>, LinkIndex> handler,
     ↪  LinkIndex left, LinkIndex right)
444  //              {
445  //                  return Links.Unsync.Each(doublet =>
446  //                  {
447  //                      var doubletIndex = doublet[Constants.IndexPart];
448  //                      if (StepRight(handler, doubletIndex, right) != Constants.Continue)
449  //                      {
450  //                          return Constants.Break;
451  //                      }
452  //                      if (left != doubletIndex)
453  //                      {
454  //                          return PartialStepRight(handler, doubletIndex, right);
455  //                      }
456  //                      return Constants.Continue;
457  //                  }, new Link<LinkIndex>(Constants.Any, Constants.Any, left));
458  //              }
459  //              [MethodImpl(MethodImplOptions.AggressiveInlining)]
460  //              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));
461  //              [MethodImpl(MethodImplOptions.AggressiveInlining)]
462  //              private LinkIndex TryStepRightUp(Func<IList<LinkIndex>, LinkIndex> handler, LinkIndex
     ↪  right, LinkIndex stepFrom)
463  //              {
464  //                  var upStep = stepFrom;
465  //                  var firstSource = Links.Unsync.GetTarget(upStep);
466  //                  while (firstSource != right && firstSource != upStep)
467  //                  {
468  //                      upStep = firstSource;
469  //                      firstSource = Links.Unsync.GetSource(upStep);
470  //                  }
471  //                  if (firstSource == right)
472  //                  {
473  //                      return handler(new LinkAddress<LinkIndex>(stepFrom));
474  //                  }
```

```
475  //                    return Constants.Continue;
476  //                }
477  //            [MethodImpl(MethodImplOptions.AggressiveInlining)]
478  //            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));
479  //            [MethodImpl(MethodImplOptions.AggressiveInlining)]
480  //            private LinkIndex TryStepLeftUp(Func<IList<LinkIndex>, LinkIndex> handler, LinkIndex
     ↪  left, LinkIndex stepFrom)
481  //            {
482  //                var upStep = stepFrom;
483  //                var firstTarget = Links.Unsync.GetSource(upStep);
484  //                while (firstTarget != left && firstTarget != upStep)
485  //                {
486  //                    upStep = firstTarget;
487  //                    firstTarget = Links.Unsync.GetTarget(upStep);
488  //                }
489  //                if (firstTarget == left)
490  //                {
491  //                    return handler(new LinkAddress<LinkIndex>(stepFrom));
492  //                }
493  //                return Constants.Continue;
494  //            }
495  //
496  //            #endregion
497  //
498  //            #region Update
499  //
500  //            /// <summary>
501  //            /// <para>
502  //            /// Updates the restriction.
503  //            /// </para>
504  //            /// <para></para>
505  //            /// </summary>
506  //            /// <param name="restriction">
507  //            /// <para>The restriction.</para>
508  //            /// <para></para>
509  //            /// </param>
510  //            /// <param name="substitution">
511  //            /// <para>The substitution.</para>
512  //            /// <para></para>
513  //            /// </param>
514  //            /// <returns>
515  //            /// <para>The link index</para>
516  //            /// <para></para>
517  //            /// </returns>
518  //            [MethodImpl(MethodImplOptions.AggressiveInlining)]
519  //            public LinkIndex Update(IList<LinkIndex>? restriction, IList<LinkIndex>?
     ↪  substitution, WriteHandler<LinkIndex> handler)
520  //            {
521  //                var sequence = restriction.SkipFirst();
522  //                var newSequence = substitution.SkipFirst();
523  //                if (sequence.IsNullOrEmpty() && newSequence.IsNullOrEmpty())
524  //                {
525  //                    return Constants.Null;
526  //                }
527  //                if (sequence.IsNullOrEmpty())
528  //                {
529  //                    return Create(substitution);
530  //                }
531  //                if (newSequence.IsNullOrEmpty())
532  //                {
533  //                    Delete(restriction);
534  //                    return Constants.Null;
535  //                }
536  //                return _sync.DoWrite((Func<ulong>)(() =>
537  //                {
538  //                    ILinksExtensions.EnsureLinkIsAnyOrExists<ulong>(Links,
     ↪  (IList<ulong>)sequence);
539  //                    Links.EnsureLinkExists(newSequence);
540  //                    return UpdateCore(sequence, newSequence);
541  //                }));
542  //            }
543  //            [MethodImpl(MethodImplOptions.AggressiveInlining)]
544  //            private LinkIndex UpdateCore(IList<LinkIndex> sequence, IList<LinkIndex> newSequence,
     ↪  WriteHandler<LinkIndex> handler)
545  //            {
```

```
546  //                LinkIndex bestVariant;
547  //                if (Options.EnforceSingleSequenceVersionOnWriteBasedOnNew &&
     →  !sequence.EqualTo(newSequence))
548  //                {
549  //                    bestVariant = CompactCore(newSequence);
550  //                }
551  //                else
552  //                {
553  //                    bestVariant = CreateCore(newSequence);
554  //                }
555  //                // TODO: Check all options only ones before loop execution
556  //                // Возможно нужно две версии Each, возвращающий фактические последовательности и
     →  с маркером,
557  //                // или возможно даже возвращать и тот и тот вариант. С другой стороны все
     →  варианты можно получить имея только фактические последовательности.
558  //                foreach (var variant in Each(sequence))
559  //                {
560  //                    if (variant != bestVariant)
561  //                    {
562  //                        UpdateOneCore(variant, bestVariant);
563  //                    }
564  //                }
565  //                return bestVariant;
566  //            }
567  //            [MethodImpl(MethodImplOptions.AggressiveInlining)]
568  //            private void UpdateOneCore(LinkIndex sequence, LinkIndex newSequence,
     →  WriteHandler<LinkIndex> handler)
569  //            {
570  //                if (Options.UseGarbageCollection)
571  //                {
572  //                    var sequenceElements = GetSequenceElements(sequence);
573  //                    var sequenceElementsContents = new
     →  Link<ulong>(Links.GetLink(sequenceElements));
574  //                    var sequenceLink = GetSequenceByElements(sequenceElements);
575  //                    var newSequenceElements = GetSequenceElements(newSequence);
576  //                    var newSequenceLink = GetSequenceByElements(newSequenceElements);
577  //                    if (Options.UseCascadeUpdate || CountUsages(sequence) == 0)
578  //                    {
579  //                        if (sequenceLink != Constants.Null)
580  //                        {
581  //                            Links.Unsync.MergeAndDelete(sequenceLink, newSequenceLink);
582  //                        }
583  //                        Links.Unsync.MergeAndDelete(sequenceElements, newSequenceElements);
584  //                    }
585  //                    ClearGarbage(sequenceElementsContents.Source);
586  //                    ClearGarbage(sequenceElementsContents.Target);
587  //                }
588  //                else
589  //                {
590  //                    if (Options.UseSequenceMarker)
591  //                    {
592  //                        var sequenceElements = GetSequenceElements(sequence);
593  //                        var sequenceLink = GetSequenceByElements(sequenceElements);
594  //                        var newSequenceElements = GetSequenceElements(newSequence);
595  //                        var newSequenceLink = GetSequenceByElements(newSequenceElements);
596  //                        if (Options.UseCascadeUpdate || CountUsages(sequence) == 0)
597  //                        {
598  //                            if (sequenceLink != Constants.Null)
599  //                            {
600  //                                Links.Unsync.MergeAndDelete(sequenceLink, newSequenceLink);
601  //                            }
602  //                            Links.Unsync.MergeAndDelete(sequenceElements, newSequenceElements);
603  //                        }
604  //                    }
605  //                    else
606  //                    {
607  //                        if (Options.UseCascadeUpdate || CountUsages(sequence) == 0)
608  //                        {
609  //                            Links.Unsync.MergeAndDelete(sequence, newSequence);
610  //                        }
611  //                    }
612  //                }
613  //            }
614  //
615  //            #endregion
616  //
617  //            #region Delete
```

```
//
//              /// <summary>
//              /// <para>
//              /// Deletes the restriction.
//              /// </para>
//              /// <para></para>
//              /// </summary>
//              /// <param name="restriction">
//              /// <para>The restriction.</para>
//              /// <para></para>
//              /// </param>
//              [MethodImpl(MethodImplOptions.AggressiveInlining)]
//              public void Delete(IList<LinkIndex>? restriction)
//              {
//                  _sync.DoWrite(() =>
//                  {
//                      var sequence = restriction.SkipFirst();
//                      // TODO: Check all options only ones before loop execution
//                      foreach (var linkToDelete in Each(sequence))
//                      {
//                          DeleteOneCore(linkToDelete);
//                      }
//                  });
//              }
//              [MethodImpl(MethodImplOptions.AggressiveInlining)]
//              private void DeleteOneCore(LinkIndex link)
//              {
//                  if (Options.UseGarbageCollection)
//                  {
//                      var sequenceElements = GetSequenceElements(link);
//                      var sequenceElementsContents = new
↪  Link<ulong>(Links.GetLink(sequenceElements));
//                      var sequenceLink = GetSequenceByElements(sequenceElements);
//                      if (Options.UseCascadeDelete || CountUsages(link) == 0)
//                      {
//                          if (sequenceLink != Constants.Null)
//                          {
//                              Links.Unsync.Delete(sequenceLink);
//                          }
//                          Links.Unsync.Delete(link);
//                      }
//                      ClearGarbage(sequenceElementsContents.Source);
//                      ClearGarbage(sequenceElementsContents.Target);
//                  }
//                  else
//                  {
//                      if (Options.UseSequenceMarker)
//                      {
//                          var sequenceElements = GetSequenceElements(link);
//                          var sequenceLink = GetSequenceByElements(sequenceElements);
//                          if (Options.UseCascadeDelete || CountUsages(link) == 0)
//                          {
//                              if (sequenceLink != Constants.Null)
//                              {
//                                  Links.Unsync.Delete(sequenceLink);
//                              }
//                              Links.Unsync.Delete(link);
//                          }
//                      }
//                      else
//                      {
//                          if (Options.UseCascadeDelete || CountUsages(link) == 0)
//                          {
//                              Links.Unsync.Delete(link);
//                          }
//                      }
//                  }
//              }
//
//          #endregion
//
//          #region Compactification
//
//              /// <summary>
//              /// <para>
//              /// Compacts the all.
//              /// </para>
```

```
694  //            /// <para></para>
695  //            /// </summary>
696  //            [MethodImpl(MethodImplOptions.AggressiveInlining)]
697  //            public void CompactAll()
698  //            {
699  //                _sync.DoWrite(() =>
700  //                {
701  //                    var sequences = Each((LinkAddress<LinkIndex>)Constants.Any);
702  //                    for (int i = 0; i < sequences.Count; i++)
703  //                    {
704  //                        var sequence = this.ToList(sequences[i]);
705  //                        Compact(sequence.ShiftRight());
706  //                    }
707  //                });
708  //            }
709  //
710  //            /// <remarks>
711  //            /// bestVariant можно выбирать по максимальному числу использований,
712  //            /// но балансированный позволяет гарантировать уникальность (если есть возможность,
713  //            /// гарантировать его использование в других местах).
714  //            ///
715  //            /// Получается этот метод должен игнорировать
     ↪ Options.EnforceSingleSequenceVersionOnWrite
716  //            /// </remarks>
717  //            [MethodImpl(MethodImplOptions.AggressiveInlining)]
718  //            public LinkIndex Compact(IList<LinkIndex> sequence)
719  //            {
720  //                return _sync.DoWrite(() =>
721  //                {
722  //                    if (sequence.IsNullOrEmpty())
723  //                    {
724  //                        return Constants.Null;
725  //                    }
726  //                    Links.EnsureInnerReferenceExists(sequence, nameof(sequence));
727  //                    return CompactCore(sequence);
728  //                });
729  //            }
730  //            [MethodImpl(MethodImplOptions.AggressiveInlining)]
731  //            private LinkIndex CompactCore(IList<LinkIndex> sequence) => UpdateCore(sequence,
     ↪ sequence);
732  //
733  //            #endregion
734  //
735  //            #region Garbage Collection
736  //            [MethodImpl(MethodImplOptions.AggressiveInlining)]
737  //            private bool IsGarbage(LinkIndex link) => link != Options.SequenceMarkerLink &&
     ↪ !Links.Unsync.IsPartialPoint(link) && Links.Count(Constants.Any, link) == 0;
738  //            [MethodImpl(MethodImplOptions.AggressiveInlining)]
739  //            private void ClearGarbage(LinkIndex link)
740  //            {
741  //                if (IsGarbage(link))
742  //                {
743  //                    var contents = new Link<ulong>(Links.GetLink(link));
744  //                    Links.Unsync.Delete(link);
745  //                    ClearGarbage(contents.Source);
746  //                    ClearGarbage(contents.Target);
747  //                }
748  //            }
749  //
750  //            #endregion
751  //
752  //            #region Walkers
753  //
754  //            /// <summary>
755  //            /// <para>
756  //            /// Determines whether this instance each part.
757  //            /// </para>
758  //            /// <para></para>
759  //            /// </summary>
760  //            /// <param name="handler">
761  //            /// <para>The handler.</para>
762  //            /// <para></para>
763  //            /// </param>
764  //            /// <param name="sequence">
765  //            /// <para>The sequence.</para>
766  //            /// <para></para>
767  //            /// </param>
```

```
768  //           /// <returns>
769  //           /// <para>The bool</para>
770  //           /// <para></para>
771  //           /// </returns>
772  //           [MethodImpl(MethodImplOptions.AggressiveInlining)]
773  //           public bool EachPart(Func<LinkIndex, bool> handler, LinkIndex sequence)
774  //           {
775  //               return _sync.DoRead(() =>
776  //               {
777  //                   var links = Links.Unsync;
778  //                   foreach (var part in Options.Walker.Walk(sequence))
779  //                   {
780  //                       if (!handler(part))
781  //                       {
782  //                           return false;
783  //                       }
784  //                   }
785  //                   return true;
786  //               });
787  //           }
788  //
789  //           /// <summary>
790  //           /// <para>
791  //           /// Represents the matcher.
792  //           /// </para>
793  //           /// <para></para>
794  //           /// </summary>
795  //           /// <seealso cref="RightSequenceWalker{LinkIndex}"/>
796  //           public class Matcher : RightSequenceWalker<LinkIndex>
797  //           {
798  //               private readonly Sequences _sequences;
799  //               private readonly IList<LinkIndex> _patternSequence;
800  //               private readonly HashSet<LinkIndex> _linksInSequence;
801  //               private readonly HashSet<LinkIndex> _results;
802  //               private readonly ReadHandler<LinkIndex> _stopableHandler;
803  //               private readonly HashSet<LinkIndex> _readAsElements;
804  //               private int _filterPosition;
805  //
806  //               /// <summary>
807  //               /// <para>
808  //               /// Initializes a new <see cref="Matcher"/> instance.
809  //               /// </para>
810  //               /// <para></para>
811  //               /// </summary>
812  //               /// <param name="sequences">
813  //               /// <para>A sequences.</para>
814  //               /// <para></para>
815  //               /// </param>
816  //               /// <param name="patternSequence">
817  //               /// <para>A pattern sequence.</para>
818  //               /// <para></para>
819  //               /// </param>
820  //               /// <param name="results">
821  //               /// <para>A results.</para>
822  //               /// <para></para>
823  //               /// </param>
824  //               /// <param name="stopableHandler">
825  //               /// <para>A stopable handler.</para>
826  //               /// <para></para>
827  //               /// </param>
828  //               /// <param name="readAsElements">
829  //               /// <para>A read as elements.</para>
830  //               /// <para></para>
831  //               /// </param>
832  //               [MethodImpl(MethodImplOptions.AggressiveInlining)]
833  //               public Matcher(Sequences sequences, IList<LinkIndex> patternSequence,
     ↪   HashSet<LinkIndex> results, ReadHandler<LinkIndex> stopableHandler, HashSet<LinkIndex>
     ↪   readAsElements = null)
834  //                   : base(sequences.Links.Unsync, new DefaultStack<LinkIndex>())
835  //               {
836  //                   _sequences = sequences;
837  //                   _patternSequence = patternSequence;
838  //                   _linksInSequence = new HashSet<LinkIndex>(patternSequence.Where(x => x !=
     ↪   _links.Constants.Any && x != ZeroOrMany));
839  //                   _results = results;
840  //                   _stopableHandler = stopableHandler;
841  //                   _readAsElements = readAsElements;
```

```
842  //                    }
843  //
844  //                    /// <summary>
845  //                    /// <para>
846  //                    /// Determines whether this instance is element.
847  //                    /// </para>
848  //                    /// <para></para>
849  //                    /// </summary>
850  //                    /// <param name="link">
851  //                    /// <para>The link.</para>
852  //                    /// <para></para>
853  //                    /// </param>
854  //                    /// <returns>
855  //                    /// <para>The bool</para>
856  //                    /// <para></para>
857  //                    /// </returns>
858  //                    [MethodImpl(MethodImplOptions.AggressiveInlining)]
859  //                    protected override bool IsElement(LinkIndex link) => base.IsElement(link) ||
     ↪   (_readAsElements != null && _readAsElements.Contains(link)) ||
     ↪   _linksInSequence.Contains(link);
860  //
861  //                    /// <summary>
862  //                    /// <para>
863  //                    /// Determines whether this instance full match.
864  //                    /// </para>
865  //                    /// <para></para>
866  //                    /// </summary>
867  //                    /// <param name="sequenceToMatch">
868  //                    /// <para>The sequence to match.</para>
869  //                    /// <para></para>
870  //                    /// </param>
871  //                    /// <returns>
872  //                    /// <para>The bool</para>
873  //                    /// <para></para>
874  //                    /// </returns>
875  //                    [MethodImpl(MethodImplOptions.AggressiveInlining)]
876  //                    public bool FullMatch(LinkIndex sequenceToMatch)
877  //                    {
878  //                        _filterPosition = 0;
879  //                        foreach (var part in Walk(sequenceToMatch))
880  //                        {
881  //                            if (!FullMatchCore(part))
882  //                            {
883  //                                break;
884  //                            }
885  //                        }
886  //                        return _filterPosition == _patternSequence.Count;
887  //                    }
888  //                    [MethodImpl(MethodImplOptions.AggressiveInlining)]
889  //                    private bool FullMatchCore(LinkIndex element)
890  //                    {
891  //                        if (_filterPosition == _patternSequence.Count)
892  //                        {
893  //                            _filterPosition = -2; // Длиннее чем нужно
894  //                            return false;
895  //                        }
896  //                        if (_patternSequence[_filterPosition] != _links.Constants.Any
897  //                         && element != _patternSequence[_filterPosition])
898  //                        {
899  //                            _filterPosition = -1;
900  //                            return false; // Начинается/Продолжается иначе
901  //                        }
902  //                        _filterPosition++;
903  //                        return true;
904  //                    }
905  //
906  //                    /// <summary>
907  //                    /// <para>
908  //                    /// Adds the full matched to results using the specified restriction.
909  //                    /// </para>
910  //                    /// <para></para>
911  //                    /// </summary>
912  //                    /// <param name="restriction">
913  //                    /// <para>The restriction.</para>
914  //                    /// <para></para>
915  //                    /// </param>
916  //                    [MethodImpl(MethodImplOptions.AggressiveInlining)]
917  //                    public void AddFullMatchedToResults(IList<LinkIndex>? restriction)
```

```
918  //                      {
919  //                          var sequenceToMatch = restriction[_links.Constants.IndexPart];
920  //                          if (FullMatch(sequenceToMatch))
921  //                          {
922  //                              _results.Add(sequenceToMatch);
923  //                          }
924  //                      }
925  //
926  //                      /// <summary>
927  //                      /// <para>
928  //                      /// Handles the full matched using the specified restriction.
929  //                      /// </para>
930  //                      /// <para></para>
931  //                      /// </summary>
932  //                      /// <param name="restriction">
933  //                      /// <para>The restriction.</para>
934  //                      /// <para></para>
935  //                      /// </param>
936  //                      /// <returns>
937  //                      /// <para>The link index</para>
938  //                      /// <para></para>
939  //                      /// </returns>
940  //                      [MethodImpl(MethodImplOptions.AggressiveInlining)]
941  //                      public LinkIndex HandleFullMatched(IList<LinkIndex>? restriction)
942  //                      {
943  //                          var sequenceToMatch = restriction[_links.Constants.IndexPart];
944  //                          if (FullMatch(sequenceToMatch) && _results.Add(sequenceToMatch))
945  //                          {
946  //                              return _stopableHandler(new LinkAddress<LinkIndex>(sequenceToMatch));
947  //                          }
948  //                          return _links.Constants.Continue;
949  //                      }
950  //
951  //                      /// <summary>
952  //                      /// <para>
953  //                      /// Handles the full matched sequence using the specified restriction.
954  //                      /// </para>
955  //                      /// <para></para>
956  //                      /// </summary>
957  //                      /// <param name="restriction">
958  //                      /// <para>The restriction.</para>
959  //                      /// <para></para>
960  //                      /// </param>
961  //                      /// <returns>
962  //                      /// <para>The link index</para>
963  //                      /// <para></para>
964  //                      /// </returns>
965  //                      [MethodImpl(MethodImplOptions.AggressiveInlining)]
966  //                      public LinkIndex HandleFullMatchedSequence(IList<LinkIndex>? restriction)
967  //                      {
968  //                          var sequenceToMatch = restriction[_links.Constants.IndexPart];
969  //                          var sequence = _sequences.GetSequenceByElements(sequenceToMatch);
970  //                          if (sequence != _links.Constants.Null && FullMatch(sequenceToMatch) &&
     ↪   _results.Add(sequenceToMatch))
971  //                          {
972  //                              return _stopableHandler(new LinkAddress<LinkIndex>(sequence));
973  //                          }
974  //                          return _links.Constants.Continue;
975  //                      }
976  //
977  //                      /// <remarks>
978  //                      /// TODO: Add support for LinksConstants.Any
979  //                      /// </remarks>
980  //                      [MethodImpl(MethodImplOptions.AggressiveInlining)]
981  //                      public bool PartialMatch(LinkIndex sequenceToMatch)
982  //                      {
983  //                          _filterPosition = -1;
984  //                          foreach (var part in Walk(sequenceToMatch))
985  //                          {
986  //                              if (!PartialMatchCore(part))
987  //                              {
988  //                                  break;
989  //                              }
990  //                          }
991  //                          return _filterPosition == _patternSequence.Count - 1;
992  //                      }
993  //                      [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
994   //                    private bool PartialMatchCore(LinkIndex element)
995   //                    {
996   //                        if (_filterPosition == (_patternSequence.Count - 1))
997   //                        {
998   //                            return false; // Нашлось
999   //                        }
1000  //                        if (_filterPosition >= 0)
1001  //                        {
1002  //                            if (element == _patternSequence[_filterPosition + 1])
1003  //                            {
1004  //                                _filterPosition++;
1005  //                            }
1006  //                            else
1007  //                            {
1008  //                                _filterPosition = -1;
1009  //                            }
1010  //                        }
1011  //                        if (_filterPosition < 0)
1012  //                        {
1013  //                            if (element == _patternSequence[0])
1014  //                            {
1015  //                                _filterPosition = 0;
1016  //                            }
1017  //                        }
1018  //                        return true; // Ищем дальше
1019  //                    }
1020  //
1021  //                    /// <summary>
1022  //                    /// <para>
1023  //                    /// Adds the partial matched to results using the specified sequence to match.
1024  //                    /// </para>
1025  //                    /// <para></para>
1026  //                    /// </summary>
1027  //                    /// <param name="sequenceToMatch">
1028  //                    /// <para>The sequence to match.</para>
1029  //                    /// <para></para>
1030  //                    /// </param>
1031  //                    [MethodImpl(MethodImplOptions.AggressiveInlining)]
1032  //                    public void AddPartialMatchedToResults(LinkIndex sequenceToMatch)
1033  //                    {
1034  //                        if (PartialMatch(sequenceToMatch))
1035  //                        {
1036  //                            _results.Add(sequenceToMatch);
1037  //                        }
1038  //                    }
1039  //
1040  //                    /// <summary>
1041  //                    /// <para>
1042  //                    /// Handles the partial matched using the specified restriction.
1043  //                    /// </para>
1044  //                    /// <para></para>
1045  //                    /// </summary>
1046  //                    /// <param name="restriction">
1047  //                    /// <para>The restriction.</para>
1048  //                    /// <para></para>
1049  //                    /// </param>
1050  //                    /// <returns>
1051  //                    /// <para>The link index</para>
1052  //                    /// <para></para>
1053  //                    /// </returns>
1054  //                    [MethodImpl(MethodImplOptions.AggressiveInlining)]
1055  //                    public LinkIndex HandlePartialMatched(IList<LinkIndex>? restriction)
1056  //                    {
1057  //                        var sequenceToMatch = restriction[_links.Constants.IndexPart];
1058  //                        if (PartialMatch(sequenceToMatch))
1059  //                        {
1060  //                            return _stopableHandler(new LinkAddress<LinkIndex>(sequenceToMatch));
1061  //                        }
1062  //                        return _links.Constants.Continue;
1063  //                    }
1064  //
1065  //                    /// <summary>
1066  //                    /// <para>
1067  //                    /// Adds the all partial matched to results using the specified sequences to
1068  //     match.
1069  //                    /// </para>
1070  //                    /// <para></para>
1071  //                    /// </summary>
```

```
1071  //              /// <param name="sequencesToMatch">
1072  //              /// <para>The sequences to match.</para>
1073  //              /// <para></para>
1074  //              /// </param>
1075  //              [MethodImpl(MethodImplOptions.AggressiveInlining)]
1076  //              public void AddAllPartialMatchedToResults(IEnumerable<LinkIndex> sequencesToMatch)
1077  //              {
1078  //                  foreach (var sequenceToMatch in sequencesToMatch)
1079  //                  {
1080  //                      if (PartialMatch(sequenceToMatch))
1081  //                      {
1082  //                          _results.Add(sequenceToMatch);
1083  //                      }
1084  //                  }
1085  //              }
1086  //
1087  //              /// <summary>
1088  //              /// <para>
1089  //              /// Adds the all partial matched to results and read as elements using the
1090  //              ↪ specified sequences to match.
1090  //              /// </para>
1091  //              /// <para></para>
1092  //              /// </summary>
1093  //              /// <param name="sequencesToMatch">
1094  //              /// <para>The sequences to match.</para>
1095  //              /// <para></para>
1096  //              /// </param>
1097  //              [MethodImpl(MethodImplOptions.AggressiveInlining)]
1098  //              public void AddAllPartialMatchedToResultsAndReadAsElements(IEnumerable<LinkIndex>
1098  //              ↪ sequencesToMatch)
1099  //              {
1100  //                  foreach (var sequenceToMatch in sequencesToMatch)
1101  //                  {
1102  //                      if (PartialMatch(sequenceToMatch))
1103  //                      {
1104  //                          _readAsElements.Add(sequenceToMatch);
1105  //                          _results.Add(sequenceToMatch);
1106  //                      }
1107  //                  }
1108  //              }
1109  //          }
1110  //
1111  //          #endregion
1112  //      }
1113  // }
```

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

```csharp
1   using System.Collections.Generic;
2   using System.Runtime.CompilerServices;
3   using Platform.Collections.Lists;
4
5   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7   namespace Platform.Data.Doublets.Sequences
8   {
9       /// <summary>
10      /// <para>
11      /// Represents the sequences extensions.
12      /// </para>
13      /// <para></para>
14      /// </summary>
15      public static class SequencesExtensions
16      {
17          /// <summary>
18          /// <para>
19          /// Creates the sequences.
20          /// </para>
21          /// <para></para>
22          /// </summary>
23          /// <typeparam name="TLinkAddress">
24          /// <para>The link.</para>
25          /// <para></para>
26          /// </typeparam>
27          /// <param name="sequences">
28          /// <para>The sequences.</para>
29          /// <para></para>
30          /// </param>
31          /// <param name="groupedSequence">
```

```
32      /// <para>The grouped sequence.</para>
33      /// <para></para>
34      /// </param>
35      /// <returns>
36      /// <para>The link</para>
37      /// <para></para>
38      /// </returns>
39      [MethodImpl(MethodImplOptions.AggressiveInlining)]
40      public static TLinkAddress Create<TLinkAddress>(this ILinks<TLinkAddress> sequences,
        ↪  IList<TLinkAddress[]> groupedSequence)
41      {
42          var finalSequence = new TLinkAddress[groupedSequence.Count];
43          for (var i = 0; i < finalSequence.Length; i++)
44          {
45              var part = groupedSequence[i];
46              finalSequence[i] = part.Length == 1 ? part[0] :
                ↪  sequences.Create(part.ShiftRight());
47          }
48          return sequences.Create(finalSequence.ShiftRight());
49      }
50
51      /// <summary>
52      /// <para>
53      /// Returns the list using the specified sequences.
54      /// </para>
55      /// <para></para>
56      /// </summary>
57      /// <typeparam name="TLinkAddress">
58      /// <para>The link.</para>
59      /// <para></para>
60      /// </typeparam>
61      /// <param name="sequences">
62      /// <para>The sequences.</para>
63      /// <para></para>
64      /// </param>
65      /// <param name="sequence">
66      /// <para>The sequence.</para>
67      /// <para></para>
68      /// </param>
69      /// <returns>
70      /// <para>The list.</para>
71      /// <para></para>
72      /// </returns>
73      [MethodImpl(MethodImplOptions.AggressiveInlining)]
74      public static IList<TLinkAddress> ToList<TLinkAddress>(this ILinks<TLinkAddress>
        ↪  sequences, TLinkAddress sequence)
75      {
76          var list = new List<TLinkAddress>();
77          var filler = new ListFiller<TLinkAddress, TLinkAddress>(list,
            ↪  sequences.Constants.Break);
78          sequences.Each(filler.AddSkipFirstAndReturnConstant, new
            ↪  LinkAddress<TLinkAddress>(sequence));
79          return list;
80      }
81  }
82 }
```

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

```
1   using System;
2   using System.Collections.Generic;
3   using Platform.Interfaces;
4   using Platform.Collections.Stacks;
5   using Platform.Converters;
6   using Platform.Data.Doublets.Sequences.Frequencies.Cache;
7   using Platform.Data.Doublets.Sequences.Frequencies.Counters;
8   using Platform.Data.Doublets.Sequences.Converters;
9   using Platform.Data.Doublets.Sequences.Walkers;
10  using Platform.Data.Doublets.Sequences.Indexes;
11  using Platform.Data.Doublets.Sequences.CriterionMatchers;
12  using System.Runtime.CompilerServices;
13
14  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
15
16  namespace Platform.Data.Doublets.Sequences
17  {
18      /// <summary>
19      /// <para>
20      /// Represents the sequences options.
21      /// </para>
```

```csharp
        /// <para></para>
        /// </summary>
        public class SequencesOptions<TLinkAddress> // TODO: To use type parameter <TLinkAddress>
        ↪  the ILinks<TLinkAddress> must contain GetConstants function.
        {
            private static readonly EqualityComparer<TLinkAddress> _equalityComparer =
            ↪  EqualityComparer<TLinkAddress>.Default;

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

```csharp
        /// <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<TLinkAddress> 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<TLinkAddress>, TLinkAddress> LinksToSequenceConverter
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
```

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

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

        /// <summary>
        /// <para>
        /// Gets or sets the walker value.
        /// </para>
        /// <para></para>
        /// </summary>
        public ISequenceWalker<TLinkAddress> 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<TLinkAddress> links)
        {
            if (UseSequenceMarker)
            {
                if (_equalityComparer.Equals(SequenceMarkerLink, links.Constants.Null))
                {
                    SequenceMarkerLink = links.CreatePoint();
                }
                else
                {
                    if (!links.Exists(SequenceMarkerLink))
                    {
                        var link = links.CreatePoint();
```

```
257                    if (!_equalityComparer.Equals(link, SequenceMarkerLink))
258                    {
259                        throw new InvalidOperationException("Cannot recreate sequence marker
     ↪    link.");
260                    }
261                }
262            }
263            if (MarkedSequenceMatcher == null)
264            {
265                MarkedSequenceMatcher = new
     ↪    MarkedSequenceCriterionMatcher<TLinkAddress>(links, SequenceMarkerLink);
266            }
267        }
268        var balancedVariantConverter = new BalancedVariantConverter<TLinkAddress>(links);
269        if (UseCompression)
270        {
271            if (LinksToSequenceConverter == null)
272            {
273                ICounter<TLinkAddress, TLinkAddress> totalSequenceSymbolFrequencyCounter;
274                if (UseSequenceMarker)
275                {
276                    totalSequenceSymbolFrequencyCounter = new
     ↪    TotalMarkedSequenceSymbolFrequencyCounter<TLinkAddress>(links,
     ↪    MarkedSequenceMatcher);
277                }
278                else
279                {
280                    totalSequenceSymbolFrequencyCounter = new
     ↪    TotalSequenceSymbolFrequencyCounter<TLinkAddress>(links);
281                }
282                var doubletFrequenciesCache = new LinkFrequenciesCache<TLinkAddress>(links,
     ↪    totalSequenceSymbolFrequencyCounter);
283                var compressingConverter = new CompressingConverter<TLinkAddress>(links,
     ↪    balancedVariantConverter, doubletFrequenciesCache);
284                LinksToSequenceConverter = compressingConverter;
285            }
286        }
287        else
288        {
289            if (LinksToSequenceConverter == null)
290            {
291                LinksToSequenceConverter = balancedVariantConverter;
292            }
293        }
294        if (UseIndex && Index == null)
295        {
296            Index = new SequenceIndex<TLinkAddress>(links);
297        }
298        if (Walker == null)
299        {
300            Walker = new RightSequenceWalker<TLinkAddress>(links, new
     ↪    DefaultStack<TLinkAddress>());
301        }
302    }

    /// <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.");
        }
    }
}
}
```

```csharp
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, TLinkAddress}"/>
    public class DateTimeToLongRawNumberSequenceConverter<TLinkAddress> : IConverter<DateTime,
       TLinkAddress>
    {
        private readonly IConverter<long, TLinkAddress> _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, TLinkAddress>
           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 TLinkAddress Convert(DateTime source) =>
           _int64ToLongRawNumberConverter.Convert(source.ToFileTimeUtc());
    }
}
```

```csharp
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{TLinkAddress, DateTime}"/>
    public class LongRawNumberSequenceToDateTimeConverter<TLinkAddress> :
       IConverter<TLinkAddress, DateTime>
    {
        private readonly IConverter<TLinkAddress, long> _longRawNumberConverterToInt64;

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

```
22          /// Initializes a new <see cref="LongRawNumberSequenceToDateTimeConverter"/> instance.
23          /// </para>
24          /// <para></para>
25          /// </summary>
26          /// <param name="longRawNumberConverterToInt64">
27          /// <para>A long raw number converter to int 64.</para>
28          /// <para></para>
29          /// </param>
30          [MethodImpl(MethodImplOptions.AggressiveInlining)]
31          public LongRawNumberSequenceToDateTimeConverter(IConverter<TLinkAddress, long>
            ↪ longRawNumberConverterToInt64) => _longRawNumberConverterToInt64 =
            ↪ longRawNumberConverterToInt64;
32
33          /// <summary>
34          /// <para>
35          /// Converts the source.
36          /// </para>
37          /// <para></para>
38          /// </summary>
39          /// <param name="source">
40          /// <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(TLinkAddress source) =>
            ↪ DateTime.FromFileTimeUtc(_longRawNumberConverterToInt64.Convert(source));
49      }
50  }
```

1.49 ./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  }
```

1.50 ./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{TLinkAddress}"/>
15      /// <seealso cref="IConverter{char, TLinkAddress}"/>
16      public class CharToUnicodeSymbolConverter<TLinkAddress> : LinksOperatorBase<TLinkAddress>,
    ↪ IConverter<char, TLinkAddress>
17      {
18          private static readonly UncheckedConverter<char, TLinkAddress> _charToAddressConverter =
        ↪ UncheckedConverter<char, TLinkAddress>.Default;
19          private readonly IConverter<TLinkAddress> _addressToNumberConverter;
20          private readonly TLinkAddress _unicodeSymbolMarker;
21
22          /// <summary>
23          /// <para>
24          /// Initializes a new <see cref="CharToUnicodeSymbolConverter"/> instance.
25          /// </para>
26          /// <para></para>
27          /// </summary>
28          /// <param name="links">
29          /// <para>A links.</para>
30          /// <para></para>
31          /// </param>
32          /// <param name="addressToNumberConverter">
33          /// <para>A address to number converter.</para>
34          /// <para></para>
35          /// </param>
36          /// <param name="unicodeSymbolMarker">
37          /// <para>A unicode symbol marker.</para>
38          /// <para></para>
39          /// </param>
40          [MethodImpl(MethodImplOptions.AggressiveInlining)]
41          public CharToUnicodeSymbolConverter(ILinks<TLinkAddress> links, IConverter<TLinkAddress>
        ↪ addressToNumberConverter, TLinkAddress unicodeSymbolMarker) : base(links)
42          {
43              _addressToNumberConverter = addressToNumberConverter;
44              _unicodeSymbolMarker = unicodeSymbolMarker;
45          }
46
47          /// <summary>
48          /// <para>
49          /// Converts the source.
50          /// </para>
51          /// <para></para>
52          /// </summary>
53          /// <param name="source">
54          /// <para>The source.</para>
55          /// <para></para>
56          /// </param>
57          /// <returns>
58          /// <para>The link</para>
59          /// <para></para>
60          /// </returns>
61          [MethodImpl(MethodImplOptions.AggressiveInlining)]
62          public TLinkAddress Convert(char source)
63          {
64              var unaryNumber =
                ↪ _addressToNumberConverter.Convert(_charToAddressConverter.Convert(source));
65              return _links.GetOrCreate(unaryNumber, _unicodeSymbolMarker);
66          }
67      }
68  }
```

1.51 ./csharp/Platform.Data.Doublets.Sequences/Unicode/StringToUnicodeSequenceConverter.cs

```
1   using System.Collections.Generic;
2   using System.Runtime.CompilerServices;
3   using Platform.Converters;
4   using Platform.Data.Doublets.Sequences.Indexes;
5
6   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8   namespace Platform.Data.Doublets.Unicode
9   {
10      /// <summary>
11      /// <para>
12      /// Represents the string to unicode sequence converter.
13      /// </para>
14      /// <para></para>
15      /// </summary>
```

```csharp
        /// <seealso cref="LinksOperatorBase{TLinkAddress}"/>
        /// <seealso cref="IConverter{string, TLinkAddress}"/>
        public class StringToUnicodeSequenceConverter<TLinkAddress> :
        ↪ LinksOperatorBase<TLinkAddress>, IConverter<string, TLinkAddress>
        {
            private readonly IConverter<string, IList<TLinkAddress>>
            ↪ _stringToUnicodeSymbolListConverter;
            private readonly IConverter<IList<TLinkAddress>, TLinkAddress>
            ↪ _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<TLinkAddress> links, IConverter<string,
            ↪ IList<TLinkAddress>> stringToUnicodeSymbolListConverter,
            ↪ IConverter<IList<TLinkAddress>, TLinkAddress> 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<TLinkAddress> links, IConverter<string,
            ↪ IList<TLinkAddress>> stringToUnicodeSymbolListConverter,
            ↪ ISequenceIndex<TLinkAddress> index, IConverter<IList<TLinkAddress>, TLinkAddress>
            ↪ listToSequenceLinkConverter, TLinkAddress unicodeSequenceMarker)
                : this(links, stringToUnicodeSymbolListConverter, new
                ↪ UnicodeSymbolsListToUnicodeSequenceConverter<TLinkAddress>(links, index,
                ↪ listToSequenceLinkConverter, unicodeSequenceMarker)) { }

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

```csharp
 83        /// </summary>
 84        /// <param name="links">
 85        /// <para>A links.</para>
 86        /// <para></para>
 87        /// </param>
 88        /// <param name="charToUnicodeSymbolConverter">
 89        /// <para>A char to unicode symbol converter.</para>
 90        /// <para></para>
 91        /// </param>
 92        /// <param name="index">
 93        /// <para>A index.</para>
 94        /// <para></para>
 95        /// </param>
 96        /// <param name="listToSequenceLinkConverter">
 97        /// <para>A list to sequence link converter.</para>
 98        /// <para></para>
 99        /// </param>
100        /// <param name="unicodeSequenceMarker">
101        /// <para>A unicode sequence marker.</para>
102        /// <para></para>
103        /// </param>
104        [MethodImpl(MethodImplOptions.AggressiveInlining)]
105        public StringToUnicodeSequenceConverter(ILinks<TLinkAddress> links, IConverter<char,
           ↪  TLinkAddress> charToUnicodeSymbolConverter, ISequenceIndex<TLinkAddress> index,
           ↪  IConverter<IList<TLinkAddress>, TLinkAddress> listToSequenceLinkConverter,
           ↪  TLinkAddress unicodeSequenceMarker)
106            : this(links, new
               ↪  StringToUnicodeSymbolsListConverter<TLinkAddress>(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<TLinkAddress> links, IConverter<char,
           ↪  TLinkAddress> charToUnicodeSymbolConverter, IConverter<IList<TLinkAddress>,
           ↪  TLinkAddress> listToSequenceLinkConverter, TLinkAddress unicodeSequenceMarker)
132            : this(links, charToUnicodeSymbolConverter, new Unindex<TLinkAddress>(),
               ↪  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<TLinkAddress> links, IConverter<string,
     ↪   IList<TLinkAddress>> stringToUnicodeSymbolListConverter,
     ↪   IConverter<IList<TLinkAddress>, TLinkAddress> listToSequenceLinkConverter,
     ↪   TLinkAddress unicodeSequenceMarker)
158            : this(links, stringToUnicodeSymbolListConverter, new Unindex<TLinkAddress>(),
     ↪   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>
172        /// <para></para>
173        /// </returns>
174        [MethodImpl(MethodImplOptions.AggressiveInlining)]
175        public TLinkAddress Convert(string source)
176        {
177            var elements = _stringToUnicodeSymbolListConverter.Convert(source);
178            return _unicodeSymbolListToSequenceConverter.Convert(elements);
179        }
180    }
181 }
```

1.52 ./csharp/Platform.Data.Doublets.Sequences/Unicode/StringToUnicodeSymbolsListConverter.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.Unicode
8  {
9      /// <summary>
10     /// <para>
11     /// Represents the string to unicode symbols list converter.
12     /// </para>
13     /// <para></para>
14     /// </summary>
15     /// <seealso cref="IConverter{string, IList{TLinkAddress}}"/>
16     public class StringToUnicodeSymbolsListConverter<TLinkAddress> : IConverter<string,
     ↪   IList<TLinkAddress>>
17     {
18         private readonly IConverter<char, TLinkAddress> _charToUnicodeSymbolConverter;
19
20         /// <summary>
21         /// <para>
22         /// Initializes a new <see cref="StringToUnicodeSymbolsListConverter"/> instance.
23         /// </para>
24         /// <para></para>
25         /// </summary>
26         /// <param name="charToUnicodeSymbolConverter">
27         /// <para>A char to unicode symbol converter.</para>
28         /// <para></para>
29         /// </param>
30         [MethodImpl(MethodImplOptions.AggressiveInlining)]
31         public StringToUnicodeSymbolsListConverter(IConverter<char, TLinkAddress>
     ↪   charToUnicodeSymbolConverter) => _charToUnicodeSymbolConverter =
     ↪   charToUnicodeSymbolConverter;
32
33         /// <summary>
34         /// <para>
35         /// Converts the source.
36         /// </para>
37         /// <para></para>
38         /// </summary>
39         /// <param name="source">
40         /// <para>The source.</para>
```

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

1.53 ./csharp/Platform.Data.Doublets.Sequences/Unicode/UnicodeMap.cs

```csharp
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.
```

```csharp
        /// </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>
```

```csharp
136        [MethodImpl(MethodImplOptions.AggressiveInlining)]
137        public static ulong FromCharToLink(char character) => (ulong)character + 1;
138
139        /// <summary>
140        /// <para>
141        /// Creates the link to char using the specified link.
142        /// </para>
143        /// <para></para>
144        /// </summary>
145        /// <param name="link">
146        /// <para>The link.</para>
147        /// <para></para>
148        /// </param>
149        /// <returns>
150        /// <para>The char</para>
151        /// <para></para>
152        /// </returns>
153        [MethodImpl(MethodImplOptions.AggressiveInlining)]
154        public static char FromLinkToChar(ulong link) => (char)(link - 1);
155
156        /// <summary>
157        /// <para>
158        /// Determines whether is char link.
159        /// </para>
160        /// <para></para>
161        /// </summary>
162        /// <param name="link">
163        /// <para>The link.</para>
164        /// <para></para>
165        /// </param>
166        /// <returns>
167        /// <para>The bool</para>
168        /// <para></para>
169        /// </returns>
170        [MethodImpl(MethodImplOptions.AggressiveInlining)]
171        public static bool IsCharLink(ulong link) => link <= MapSize;
172
173        /// <summary>
174        /// <para>
175        /// Creates the links to string using the specified links list.
176        /// </para>
177        /// <para></para>
178        /// </summary>
179        /// <param name="linksList">
180        /// <para>The links list.</para>
181        /// <para></para>
182        /// </param>
183        /// <returns>
184        /// <para>The string</para>
185        /// <para></para>
186        /// </returns>
187        [MethodImpl(MethodImplOptions.AggressiveInlining)]
188        public static string FromLinksToString(IList<ulong> linksList)
189        {
190            var sb = new StringBuilder();
191            for (int i = 0; i < linksList.Count; i++)
192            {
193                sb.Append(FromLinkToChar(linksList[i]));
194            }
195            return sb.ToString();
196        }
197
198        /// <summary>
199        /// <para>
200        /// Creates the sequence link to string using the specified link.
201        /// </para>
202        /// <para></para>
203        /// </summary>
204        /// <param name="link">
205        /// <para>The link.</para>
206        /// <para></para>
207        /// </param>
208        /// <param name="links">
209        /// <para>The links.</para>
210        /// <para></para>
211        /// </param>
212        /// <returns>
213        /// <para>The string</para>
```

```csharp
        /// <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>
```

```csharp
        /// <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)
```

```
368                {
369                    var relativeLength = 1;
370                    if (array[offset] <= LastCharLink)
371                    {
372                        var currentCategory =
                             ↪  CharUnicodeInfo.GetUnicodeCategory(FromLinkToChar(array[offset]));
373                        var absoluteLength = offset + relativeLength;
374                        while (absoluteLength < array.Length &&
375                               array[absoluteLength] <= LastCharLink &&
376                               currentCategory == CharUnicodeInfo.GetUnicodeCategory(FromLinkToChar(
                               ↪  array[absoluteLength])))
377                        {
378                            relativeLength++;
379                            absoluteLength++;
380                        }
381                    }
382                    else
383                    {
384                        var absoluteLength = offset + relativeLength;
385                        while (absoluteLength < array.Length && array[absoluteLength] > LastCharLink)
386                        {
387                            relativeLength++;
388                            absoluteLength++;
389                        }
390                    }
391                    // copy array
392                    var innerSequence = new ulong[relativeLength];
393                    var maxLength = offset + relativeLength;
394                    for (var i = offset; i < maxLength; i++)
395                    {
396                        innerSequence[i - offset] = array[i];
397                    }
398                    result.Add(innerSequence);
399                    offset += relativeLength;
400                }
401                return result;
402            }
403        }
404    }
```

1.54 ./csharp/Platform.Data.Doublets.Sequences/Unicode/UnicodeSequenceToStringConverter.cs

```
1   using System;
2   using System.Collections.Generic;
3   using System.Runtime.CompilerServices;
4   using Platform.Interfaces;
5   using Platform.Converters;
6   using Platform.Data.Doublets.Sequences.Walkers;
7   using System.Text;
8   using Platform.Data.Doublets.Sequences.CriterionMatchers;
9
10  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
11
12  namespace Platform.Data.Doublets.Unicode
13  {
14      /// <summary>
15      /// <para>
16      /// Represents the unicode sequence to string converter.
17      /// </para>
18      /// <para></para>
19      /// </summary>
20      /// <seealso cref="LinksOperatorBase{TLinkAddress}"/>
21      /// <seealso cref="IConverter{TLinkAddress, string}"/>
22      public class UnicodeSequenceToStringConverter<TLinkAddress> :
          ↪  LinksOperatorBase<TLinkAddress>, IConverter<TLinkAddress, string>
23      {
24          private readonly ICriterionMatcher<TLinkAddress> _unicodeSequenceCriterionMatcher;
25          private readonly ISequenceWalker<TLinkAddress> _sequenceWalker;
26          private readonly IConverter<TLinkAddress, char> _unicodeSymbolToCharConverter;
27          private readonly TLinkAddress _unicodeSequenceMarker;
28
29
30          /// <summary>
31          /// <para>
32          /// Initializes a new <see cref="UnicodeSequenceToStringConverter"/> instance.
33          /// </para>
34          /// <para></para>
35          /// </summary>
36          /// <param name="links">
37          /// <para>A links.</para>
38          /// <para></para>
```

```csharp
        /// </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<TLinkAddress> links,
        ↪   ICriterionMatcher<TLinkAddress> unicodeSequenceCriterionMatcher,
        ↪   ISequenceWalker<TLinkAddress> sequenceWalker, IConverter<TLinkAddress, char>
        ↪   unicodeSymbolToCharConverter, TLinkAddress unicodeSequenceMarker) : base(links)
        {
            _unicodeSequenceCriterionMatcher = unicodeSequenceCriterionMatcher;
            _sequenceWalker = sequenceWalker;
            _unicodeSymbolToCharConverter = unicodeSymbolToCharConverter;
            _unicodeSequenceMarker = unicodeSequenceMarker;
        }

        public UnicodeSequenceToStringConverter(ILinks<TLinkAddress> links,
        ↪   ISequenceWalker<TLinkAddress> sequenceWalker, IConverter<TLinkAddress, char>
        ↪   unicodeSymbolToCharConverter, TLinkAddress unicodeSequenceMarker): this(links, new
        ↪   UnicodeSequenceMatcher<TLinkAddress>(links, unicodeSequenceMarker), sequenceWalker,
        ↪   unicodeSymbolToCharConverter, unicodeSequenceMarker){}


        /// <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(TLinkAddress source)
        {
            if (!_unicodeSequenceCriterionMatcher.IsMatched(source))
            {
                throw new ArgumentOutOfRangeException(nameof(source), source, "Specified link is
                ↪   not a unicode sequence.");
            }
            if(EqualityComparer<TLinkAddress>.Default.Equals(_unicodeSequenceMarker, source))
            {
                return String.Empty;
            }
            var sequence = _links.GetSource(source);
            var sb = new StringBuilder();
            foreach(var character in _sequenceWalker.Walk(sequence))
            {
                sb.Append(_unicodeSymbolToCharConverter.Convert(character));
            }
            return sb.ToString();
        }
    }
}
```

1.55   ./csharp/Platform.Data.Doublets.Sequences/Unicode/UnicodeSymbolToCharConverter.cs

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

```csharp
#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{TLinkAddress}"/>
    /// <seealso cref="IConverter{TLinkAddress, char}"/>
    public class UnicodeSymbolToCharConverter<TLinkAddress> : LinksOperatorBase<TLinkAddress>,
        IConverter<TLinkAddress, char>
    {
        private static readonly UncheckedConverter<TLinkAddress, char> _addressToCharConverter =
            UncheckedConverter<TLinkAddress, char>.Default;
        private readonly IConverter<TLinkAddress> _numberToAddressConverter;
        private readonly ICriterionMatcher<TLinkAddress> _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<TLinkAddress> links, IConverter<TLinkAddress>
            numberToAddressConverter, ICriterionMatcher<TLinkAddress>
            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(TLinkAddress 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)));
        }
    }
}
```

```
77        }

1.56    ./csharp/Platform.Data.Doublets.Sequences/Unicode/UnicodeSymbolsListToUnicodeSequenceConverter.cs
  1    using System.Collections.Generic;
  2    using System.Runtime.CompilerServices;
  3    using Platform.Collections;
  4    using Platform.Converters;
  5    using Platform.Data.Doublets.Sequences.Indexes;
  6
  7    #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
  8
  9    namespace Platform.Data.Doublets.Unicode
 10    {
 11        /// <summary>
 12        /// <para>
 13        /// Represents the unicode symbols list to unicode sequence converter.
 14        /// </para>
 15        /// <para></para>
 16        /// </summary>
 17        /// <seealso cref="LinksOperatorBase{TLinkAddress}"/>
 18        /// <seealso cref="IConverter{IList{TLinkAddress}, TLinkAddress}"/>
 19        public class UnicodeSymbolsListToUnicodeSequenceConverter<TLinkAddress> :
 ↪          LinksOperatorBase<TLinkAddress>, IConverter<IList<TLinkAddress>, TLinkAddress>
 20        {
 21            private readonly ISequenceIndex<TLinkAddress> _index;
 22            private readonly IConverter<IList<TLinkAddress>, TLinkAddress>
 ↪              _listToSequenceLinkConverter;
 23            private readonly TLinkAddress _unicodeSequenceMarker;
 24
 25            /// <summary>
 26            /// <para>
 27            /// Initializes a new <see cref="UnicodeSymbolsListToUnicodeSequenceConverter"/>
 ↪              instance.
 28            /// </para>
 29            /// <para></para>
 30            /// </summary>
 31            /// <param name="links">
 32            /// <para>A links.</para>
 33            /// <para></para>
 34            /// </param>
 35            /// <param name="index">
 36            /// <para>A index.</para>
 37            /// <para></para>
 38            /// </param>
 39            /// <param name="listToSequenceLinkConverter">
 40            /// <para>A list to sequence link converter.</para>
 41            /// <para></para>
 42            /// </param>
 43            /// <param name="unicodeSequenceMarker">
 44            /// <para>A unicode sequence marker.</para>
 45            /// <para></para>
 46            /// </param>
 47            [MethodImpl(MethodImplOptions.AggressiveInlining)]
 48            public UnicodeSymbolsListToUnicodeSequenceConverter(ILinks<TLinkAddress> links,
 ↪              ISequenceIndex<TLinkAddress> index, IConverter<IList<TLinkAddress>, TLinkAddress>
 ↪              listToSequenceLinkConverter, TLinkAddress unicodeSequenceMarker) : base(links)
 49            {
 50                _index = index;
 51                _listToSequenceLinkConverter = listToSequenceLinkConverter;
 52                _unicodeSequenceMarker = unicodeSequenceMarker;
 53            }
 54
 55            /// <summary>
 56            /// <para>
 57            /// Initializes a new <see cref="UnicodeSymbolsListToUnicodeSequenceConverter"/>
 ↪              instance.
 58            /// </para>
 59            /// <para></para>
 60            /// </summary>
 61            /// <param name="links">
 62            /// <para>A links.</para>
 63            /// <para></para>
 64            /// </param>
 65            /// <param name="listToSequenceLinkConverter">
 66            /// <para>A list to sequence link converter.</para>
 67            /// <para></para>
 68            /// </param>
 69            /// <param name="unicodeSequenceMarker">
 70            /// <para>A unicode sequence marker.</para>
```

```csharp
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public UnicodeSymbolsListToUnicodeSequenceConverter(ILinks<TLinkAddress> links,
            IConverter<IList<TLinkAddress>, TLinkAddress> listToSequenceLinkConverter,
            TLinkAddress unicodeSequenceMarker)
            : this(links, new Unindex<TLinkAddress>(), 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 TLinkAddress Convert(IList<TLinkAddress> list)
        {
            if (list.IsNullOrEmpty())
            {
                return _unicodeSequenceMarker;
            }
            _index.Add(list);
            var sequence = _listToSequenceLinkConverter.Convert(list);
            return _links.GetOrCreate(sequence, _unicodeSequenceMarker);
        }
    }
}
```

1.57 ./csharp/Platform.Data.Doublets.Sequences/Walkers/ISequenceWalker.cs

```csharp
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<TLinkAddress>
    {
        /// <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<TLinkAddress> Walk(TLinkAddress sequence);
    }
}
```

1.58 ./csharp/Platform.Data.Doublets.Sequences/Walkers/LeftSequenceWalker.cs

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

```csharp
#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{TLinkAddress}"/>
    public class LeftSequenceWalker<TLinkAddress> : SequenceWalkerBase<TLinkAddress>
    {
        /// <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<TLinkAddress> links, IStack<TLinkAddress> stack,
        ↪   Func<TLinkAddress, 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<TLinkAddress> links, IStack<TLinkAddress> 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 TLinkAddress GetNextElementAfterPop(TLinkAddress element) =>
        ↪   _links.GetSource(element);

        /// <summary>
        /// <para>
        /// Gets the next element after push using the specified element.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="element">
```

```csharp
        /// <para>The element.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLinkAddress GetNextElementAfterPush(TLinkAddress 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<TLinkAddress> WalkContents(TLinkAddress 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;
                }
            }
        }
    }
}
```

1.59 ./csharp/Platform.Data.Doublets.Sequences/Walkers/LeveledSequenceWalker.cs

```csharp
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{TLinkAddress}"/>
    /// <seealso cref="ISequenceWalker{TLinkAddress}"/>
    public class LeveledSequenceWalker<TLinkAddress> : LinksOperatorBase<TLinkAddress>,
    ↪    ISequenceWalker<TLinkAddress>
    {
        private static readonly EqualityComparer<TLinkAddress> _equalityComparer =
        ↪    EqualityComparer<TLinkAddress>.Default;
        private readonly Func<TLinkAddress, bool> _isElement;

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

```
34          /// <para>A links.</para>
35          /// <para></para>
36          /// </param>
37          /// <param name="isElement">
38          /// <para>A is element.</para>
39          /// <para></para>
40          /// </param>
41          [MethodImpl(MethodImplOptions.AggressiveInlining)]
42          public LeveledSequenceWalker(ILinks<TLinkAddress> links, Func<TLinkAddress, bool>
   ↪  isElement) : base(links) => _isElement = isElement;

43
44          /// <summary>
45          /// <para>
46          /// Initializes a new <see cref="LeveledSequenceWalker"/> instance.
47          /// </para>
48          /// <para></para>
49          /// </summary>
50          /// <param name="links">
51          /// <para>A links.</para>
52          /// <para></para>
53          /// </param>
54          [MethodImpl(MethodImplOptions.AggressiveInlining)]
55          public LeveledSequenceWalker(ILinks<TLinkAddress> links) : base(links) => _isElement =
   ↪  _links.IsPartialPoint;

56
57          /// <summary>
58          /// <para>
59          /// Walks the sequence.
60          /// </para>
61          /// <para></para>
62          /// </summary>
63          /// <param name="sequence">
64          /// <para>The sequence.</para>
65          /// <para></para>
66          /// </param>
67          /// <returns>
68          /// <para>An enumerable of t link</para>
69          /// <para></para>
70          /// </returns>
71          [MethodImpl(MethodImplOptions.AggressiveInlining)]
72          public IEnumerable<TLinkAddress> Walk(TLinkAddress sequence) => ToArray(sequence);

73
74          /// <summary>
75          /// <para>
76          /// Returns the array using the specified sequence.
77          /// </para>
78          /// <para></para>
79          /// </summary>
80          /// <param name="sequence">
81          /// <para>The sequence.</para>
82          /// <para></para>
83          /// </param>
84          /// <returns>
85          /// <para>The link array</para>
86          /// <para></para>
87          /// </returns>
88          [MethodImpl(MethodImplOptions.AggressiveInlining)]
89          public TLinkAddress[] ToArray(TLinkAddress sequence)
90          {
91              var length = 1;
92              var array = new TLinkAddress[length];
93              array[0] = sequence;
94              if (_isElement(sequence))
95              {
96                  return array;
97              }
98              bool hasElements;
99              do
100             {
101                 length *= 2;
102 #if USEARRAYPOOL
103                 var nextArray = ArrayPool.Allocate<ulong>(length);
104 #else
105                 var nextArray = new TLinkAddress[length];
106 #endif
107                 hasElements = false;
108                 for (var i = 0; i < array.Length; i++)
109                 {
```

```
110                        var candidate = array[i];
111                        if (_equalityComparer.Equals(array[i], default))
112                        {
113                            continue;
114                        }
115                        var doubletOffset = i * 2;
116                        if (_isElement(candidate))
117                        {
118                            nextArray[doubletOffset] = candidate;
119                        }
120                        else
121                        {
122                            var links = _links;
123                            var link = links.GetLink(candidate);
124                            var linkSource = links.GetSource(link);
125                            var linkTarget = links.GetTarget(link);
126                            nextArray[doubletOffset] = linkSource;
127                            nextArray[doubletOffset + 1] = linkTarget;
128                            if (!hasElements)
129                            {
130                                hasElements = !(_isElement(linkSource) && _isElement(linkTarget));
131                            }
132                        }
133                    }
134 #if USEARRAYPOOL
135                    if (array.Length > 1)
136                    {
137                        ArrayPool.Free(array);
138                    }
139 #endif
140                    array = nextArray;
141                }
142                while (hasElements);
143                var filledElementsCount = CountFilledElements(array);
144                if (filledElementsCount == array.Length)
145                {
146                    return array;
147                }
148                else
149                {
150                    return CopyFilledElements(array, filledElementsCount);
151                }
152            }
153            [MethodImpl(MethodImplOptions.AggressiveInlining)]
154            private static TLinkAddress[] CopyFilledElements(TLinkAddress[] array, int
       ↪  filledElementsCount)
155            {
156                var finalArray = new TLinkAddress[filledElementsCount];
157                for (int i = 0, j = 0; i < array.Length; i++)
158                {
159                    if (!_equalityComparer.Equals(array[i], default))
160                    {
161                        finalArray[j] = array[i];
162                        j++;
163                    }
164                }
165 #if USEARRAYPOOL
166                ArrayPool.Free(array);
167 #endif
168                return finalArray;
169            }
170            [MethodImpl(MethodImplOptions.AggressiveInlining)]
171            private static int CountFilledElements(TLinkAddress[] array)
172            {
173                var count = 0;
174                for (var i = 0; i < array.Length; i++)
175                {
176                    if (!_equalityComparer.Equals(array[i], default))
177                    {
178                        count++;
179                    }
180                }
181                return count;
182            }
183        }
184 }
```

```csharp
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{TLinkAddress}"/>
    public class RightSequenceWalker<TLinkAddress> : SequenceWalkerBase<TLinkAddress>
    {
        /// <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<TLinkAddress> links, IStack<TLinkAddress> stack,
        ↪   Func<TLinkAddress, 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<TLinkAddress> links, IStack<TLinkAddress> 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 TLinkAddress GetNextElementAfterPop(TLinkAddress element) =>
        ↪   _links.GetTarget(element);

        /// <summary>
```

```csharp
        /// <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 TLinkAddress GetNextElementAfterPush(TLinkAddress 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<TLinkAddress> WalkContents(TLinkAddress 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;
                }
            }
        }
    }
}
```

1.61   ./csharp/Platform.Data.Doublets.Sequences/Walkers/SequenceWalkerBase.cs

```csharp
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{TLinkAddress}"/>
    /// <seealso cref="ISequenceWalker{TLinkAddress}"/>
    public abstract class SequenceWalkerBase<TLinkAddress> : LinksOperatorBase<TLinkAddress>,
      ↪  ISequenceWalker<TLinkAddress>
    {
        private readonly IStack<TLinkAddress> _stack;
        private readonly Func<TLinkAddress, bool> _isElement;

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

```csharp
        /// <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<TLinkAddress> links, IStack<TLinkAddress> stack,
        ↪  Func<TLinkAddress, 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<TLinkAddress> links, IStack<TLinkAddress> 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<TLinkAddress> Walk(TLinkAddress 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
                    {
```

```
107                        _stack.Push(element);
108                        element = GetNextElementAfterPush(element);
109                    }
110                }
111            }
112        }
113
114        /// <summary>
115        /// <para>
116        /// Determines whether this instance is element.
117        /// </para>
118        /// <para></para>
119        /// </summary>
120        /// <param name="elementLink">
121        /// <para>The element link.</para>
122        /// <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(TLinkAddress 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 TLinkAddress GetNextElementAfterPop(TLinkAddress 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 TLinkAddress GetNextElementAfterPush(TLinkAddress 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<TLinkAddress> WalkContents(TLinkAddress element);
181    }
182 }
```

```csharp
using System.Collections.Generic;
using System.Numerics;
using Platform.Data.Doublets.Memory;
using Platform.Data.Doublets.Memory.United.Generic;
using Platform.Data.Doublets.Numbers.Raw;
using Platform.Data.Doublets.Sequences.Converters;
using Platform.Data.Numbers.Raw;
using Platform.Memory;
using Xunit;
using TLinkAddress = System.UInt64;

namespace Platform.Data.Doublets.Sequences.Tests
{
    public class BigIntegerConvertersTests
    {
        public ILinks<TLinkAddress> CreateLinks() => CreateLinks<TLinkAddress>(new
            IO.TemporaryFile());

        public ILinks<TLinkAddress> CreateLinks<TLinkAddress>(string dataDbFilename)
        {
            var linksConstants = new
                LinksConstants<TLinkAddress>(enableExternalReferencesSupport: true);
            return new UnitedMemoryLinks<TLinkAddress>(new
                FileMappedResizableDirectMemory(dataDbFilename),
                UnitedMemoryLinks<TLinkAddress>.DefaultLinksSizeStep, linksConstants,
                IndexTreeType.Default);
        }

        [Fact]
        public void DecimalMaxValueTest()
        {
            var links = CreateLinks();
            BigInteger bigInteger = new(decimal.MaxValue);
            TLinkAddress negativeNumberMarker = links.Create();
            AddressToRawNumberConverter<TLinkAddress> addressToRawNumberConverter = new();
            RawNumberToAddressConverter<TLinkAddress> numberToAddressConverter = new();
            BalancedVariantConverter<TLinkAddress> listToSequenceConverter = new(links);
            BigIntegerToRawNumberSequenceConverter<TLinkAddress>
                bigIntegerToRawNumberSequenceConverter = new(links, addressToRawNumberConverter,
                listToSequenceConverter, negativeNumberMarker);
            RawNumberSequenceToBigIntegerConverter<TLinkAddress>
                rawNumberSequenceToBigIntegerConverter = new(links, numberToAddressConverter,
                negativeNumberMarker);
            var bigIntSequence = bigIntegerToRawNumberSequenceConverter.Convert(bigInteger);
            var bigIntFromSequence =
                rawNumberSequenceToBigIntegerConverter.Convert(bigIntSequence);
            Assert.Equal(bigInteger, bigIntFromSequence);
        }

        [Fact]
        public void DecimalMinValueTest()
        {
            var links = CreateLinks();
            BigInteger bigInteger = new(decimal.MinValue);
            TLinkAddress negativeNumberMarker = links.Create();
            AddressToRawNumberConverter<TLinkAddress> addressToRawNumberConverter = new();
            RawNumberToAddressConverter<TLinkAddress> numberToAddressConverter = new();
            BalancedVariantConverter<TLinkAddress> listToSequenceConverter = new(links);
            BigIntegerToRawNumberSequenceConverter<TLinkAddress>
                bigIntegerToRawNumberSequenceConverter = new(links, addressToRawNumberConverter,
                listToSequenceConverter, negativeNumberMarker);
            RawNumberSequenceToBigIntegerConverter<TLinkAddress>
                rawNumberSequenceToBigIntegerConverter = new(links, numberToAddressConverter,
                negativeNumberMarker);
            var bigIntSequence = bigIntegerToRawNumberSequenceConverter.Convert(bigInteger);
            var bigIntFromSequence =
                rawNumberSequenceToBigIntegerConverter.Convert(bigIntSequence);
            Assert.Equal(bigInteger, bigIntFromSequence);
        }

        [Fact]
        public void ZeroValueTest()
        {
            var links = CreateLinks();
            BigInteger bigInteger = new(0);
            TLinkAddress negativeNumberMarker = links.Create();
            AddressToRawNumberConverter<TLinkAddress> addressToRawNumberConverter = new();
```

```
63              RawNumberToAddressConverter<TLinkAddress> numberToAddressConverter = new();
64              BalancedVariantConverter<TLinkAddress> listToSequenceConverter = new(links);
65              BigIntegerToRawNumberSequenceConverter<TLinkAddress>
    ↪           bigIntegerToRawNumberSequenceConverter = new(links, addressToRawNumberConverter,
    ↪           listToSequenceConverter, negativeNumberMarker);
66              RawNumberSequenceToBigIntegerConverter<TLinkAddress>
    ↪           rawNumberSequenceToBigIntegerConverter = new(links, numberToAddressConverter,
    ↪           negativeNumberMarker);
67              var bigIntSequence = bigIntegerToRawNumberSequenceConverter.Convert(bigInteger);
68              var bigIntFromSequence =
    ↪           rawNumberSequenceToBigIntegerConverter.Convert(bigIntSequence);
69              Assert.Equal(bigInteger, bigIntFromSequence);
70          }
71
72          [Fact]
73          public void OneValueTest()
74          {
75              var links = CreateLinks();
76              BigInteger bigInteger = new(1);
77              TLinkAddress negativeNumberMarker = links.Create();
78              AddressToRawNumberConverter<TLinkAddress> addressToRawNumberConverter = new();
79              RawNumberToAddressConverter<TLinkAddress> numberToAddressConverter = new();
80              BalancedVariantConverter<TLinkAddress> listToSequenceConverter = new(links);
81              BigIntegerToRawNumberSequenceConverter<TLinkAddress>
    ↪           bigIntegerToRawNumberSequenceConverter = new(links, addressToRawNumberConverter,
    ↪           listToSequenceConverter, negativeNumberMarker);
82              RawNumberSequenceToBigIntegerConverter<TLinkAddress>
    ↪           rawNumberSequenceToBigIntegerConverter = new(links, numberToAddressConverter,
    ↪           negativeNumberMarker);
83              var bigIntSequence = bigIntegerToRawNumberSequenceConverter.Convert(bigInteger);
84              var bigIntFromSequence =
    ↪           rawNumberSequenceToBigIntegerConverter.Convert(bigIntSequence);
85              Assert.Equal(bigInteger, bigIntFromSequence);
86          }
87      }
88  }
```

1.63 ./csharp/Platform.Data.Doublets.Sequences.Tests/DefaultSequenceAppenderTests.cs

```
1   using System.Collections.Generic;
2   using Platform.Collections.Stacks;
3   using Platform.Data.Doublets.Memory;
4   using Platform.Data.Doublets.Memory.United.Generic;
5   using Platform.Data.Doublets.Sequences;
6   using Platform.Data.Doublets.Sequences.HeightProviders;
7   using Platform.Data.Numbers.Raw;
8   using Platform.Interfaces;
9   using Platform.Memory;
10  using Platform.Numbers;
11  using Xunit;
12  using Xunit.Abstractions;
13  using TLinkAddress = System.UInt64;
14
15  namespace Platform.Data.Doublets.Sequences.Tests
16  {
17      public class DefaultSequenceAppenderTests
18      {
19          private readonly ITestOutputHelper _output;
20
21          public DefaultSequenceAppenderTests(ITestOutputHelper output)
22          {
23              _output = output;
24          }
25          public static ILinks<TLinkAddress> CreateLinks() => CreateLinks<TLinkAddress>(new
    ↪       IO.TemporaryFile());
26
27          public static ILinks<TLinkAddress> CreateLinks<TLinkAddress>(string dataDBFilename)
28          {
29              var linksConstants = new
    ↪           LinksConstants<TLinkAddress>(enableExternalReferencesSupport: true);
30              return new UnitedMemoryLinks<TLinkAddress>(new
    ↪           FileMappedResizableDirectMemory(dataDBFilename),
    ↪           UnitedMemoryLinks<TLinkAddress>.DefaultLinksSizeStep, linksConstants,
    ↪           IndexTreeType.Default);
31          }
32
33          public class ValueCriterionMatcher<TLinkAddress> : ICriterionMatcher<TLinkAddress>
34          {
35              public readonly ILinks<TLinkAddress> Links;
36              public readonly TLinkAddress Marker;
```

```csharp
            public ValueCriterionMatcher(ILinks<TLinkAddress> links, TLinkAddress marker)
            {
                Links = links;
                Marker = marker;
            }

            public bool IsMatched(TLinkAddress link) =>
                EqualityComparer<TLinkAddress>.Default.Equals(Links.GetSource(link), Marker);
        }

        [Fact]
        public void AppendArrayBug()
        {
            ILinks<TLinkAddress> links = CreateLinks();
            TLinkAddress 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<TLinkAddress> valueCriterionMatcher = new(links, meaningRoot);
            DefaultSequenceRightHeightProvider<ulong> defaultSequenceRightHeightProvider =
                new(links, valueCriterionMatcher);
            DefaultSequenceAppender<TLinkAddress> 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);
        }
    }
}
```

1.64 ./csharp/Platform.Data.Doublets.Sequences.Tests/ILinksExtensionsTests.cs

```csharp
// using Xunit;
//
// namespace Platform.Data.Doublets.Sequences.Tests
// {
//     public class ILinksExtensionsTests
//     {
//         [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);
//             }
//         }
//     }
// }
```

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

```csharp
// 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
```

```csharp
// {
//     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.";
//
//         [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);
```

```
 96  //                    var unaryNumberIncrementer = new UnaryNumberIncrementer<ulong>(links,
     ↪  unaryOne);
 97  //                    var frequencyIncrementer = new FrequencyIncrementer<ulong>(links,
     ↪  frequencyMarker, unaryOne, unaryNumberIncrementer);
 98  //                    var frequencyPropertyOperator = new PropertyOperator<ulong>(links,
     ↪  frequencyPropertyMarker, frequencyMarker);
 99  //                    var index = new FrequencyIncrementingSequenceIndex<ulong>(links,
     ↪  frequencyPropertyOperator, frequencyIncrementer);
100  //                    var linkToItsFrequencyNumberConverter = new
     ↪  LinkToItsFrequencyNumberConveter<ulong>(links, frequencyPropertyOperator,
     ↪  unaryNumberToAddressConverter);
101  //                    var sequenceToItsLocalElementLevelsConverter = new
     ↪  SequenceToItsLocalElementLevelsConverter<ulong>(links, linkToItsFrequencyNumberConverter);
102  //                    var optimalVariantConverter = new OptimalVariantConverter<ulong>(links,
     ↪  sequenceToItsLocalElementLevelsConverter);
103  //
104  //                    var sequences = new Sequences(links, new SequencesOptions<ulong>() { Walker =
     ↪  new LeveledSequenceWalker<ulong>(links) });
105  //
106  //                    ExecuteTest(sequences, sequence, sequenceToItsLocalElementLevelsConverter,
     ↪  index, optimalVariantConverter);
107  //                }
108  //            }
109  //
110  //            [Fact]
111  //            public static void DictionaryBasedFrequencyStoredOptimalVariantSequenceTest()
112  //            {
113  //                using (var scope = new TempLinksTestScope(useSequences: false))
114  //                {
115  //                    var links = scope.Links;
116  //
117  //                    links.UseUnicode();
118  //
119  //                    var sequence = UnicodeMap.FromStringToLinkArray(_sequenceExample);
120  //
121  //                    var totalSequenceSymbolFrequencyCounter = new
     ↪  TotalSequenceSymbolFrequencyCounter<ulong>(links);
122  //
123  //                    var linkFrequenciesCache = new LinkFrequenciesCache<ulong>(links,
     ↪  totalSequenceSymbolFrequencyCounter);
124  //
125  //                    var index = new
     ↪  CachedFrequencyIncrementingSequenceIndex<ulong>(linkFrequenciesCache);
126  //                    var linkToItsFrequencyNumberConverter = new
     ↪  FrequenciesCacheBasedLinkToItsFrequencyNumberConverter<ulong>(linkFrequenciesCache);
127  //
128  //                    var sequenceToItsLocalElementLevelsConverter = new
     ↪  SequenceToItsLocalElementLevelsConverter<ulong>(links, linkToItsFrequencyNumberConverter);
129  //                    var optimalVariantConverter = new OptimalVariantConverter<ulong>(links,
     ↪  sequenceToItsLocalElementLevelsConverter);
130  //
131  //                    var sequences = new Sequences(links, new SequencesOptions<ulong>() { Walker =
     ↪  new LeveledSequenceWalker<ulong>(links) });
132  //
133  //                    ExecuteTest(sequences, sequence, sequenceToItsLocalElementLevelsConverter,
     ↪  index, optimalVariantConverter);
134  //                }
135  //            }
136  //            private static void ExecuteTest(Sequences sequences, ulong[] sequence,
     ↪  SequenceToItsLocalElementLevelsConverter<ulong> sequenceToItsLocalElementLevelsConverter,
     ↪  ISequenceIndex<ulong> index, OptimalVariantConverter<ulong> optimalVariantConverter)
137  //            {
138  //                index.Add(sequence);
139  //
140  //                var optimalVariant = optimalVariantConverter.Convert(sequence);
141  //
142  //                var readSequence1 = sequences.ToList(optimalVariant);
143  //
144  //                Assert.True(sequence.SequenceEqual(readSequence1));
145  //            }
146  //
147  //            [Fact]
148  //            public static void SavedSequencesOptimizationTest()
149  //            {
150  //                LinksConstants<ulong> constants = new LinksConstants<ulong>((1, long.MaxValue),
     ↪  (long.MaxValue + 1UL, ulong.MaxValue));
```

```
151  //
152  //                  using (var memory = new HeapResizableDirectMemory())
153  //                  using (var disposableLinks = new UInt64UnitedMemoryLinks(memory,
     ↪  UInt64UnitedMemoryLinks.DefaultLinksSizeStep, constants, IndexTreeType.Default))
154  //                  {
155  //                      var links = new UInt64Links(disposableLinks);
156  //
157  //                      var root = links.CreatePoint();
158  //
159  //                      //var numberToAddressConverter = new RawNumberToAddressConverter<ulong>();
160  //                      var addressToNumberConverter = new AddressToRawNumberConverter<ulong>();
161  //
162  //                      var unicodeSymbolMarker = links.GetOrCreate(root,
     ↪  addressToNumberConverter.Convert(1));
163  //                      var unicodeSequenceMarker = links.GetOrCreate(root,
     ↪  addressToNumberConverter.Convert(2));
164  //
165  //                      var totalSequenceSymbolFrequencyCounter = new
     ↪  TotalSequenceSymbolFrequencyCounter<ulong>(links);
166  //                      var linkFrequenciesCache = new LinkFrequenciesCache<ulong>(links,
     ↪  totalSequenceSymbolFrequencyCounter);
167  //                      var index = new
     ↪  CachedFrequencyIncrementingSequenceIndex<ulong>(linkFrequenciesCache);
168  //                      var linkToItsFrequencyNumberConverter = new
     ↪  FrequenciesCacheBasedLinkToItsFrequencyNumberConverter<ulong>(linkFrequenciesCache);
169  //                      var sequenceToItsLocalElementLevelsConverter = new
     ↪  SequenceToItsLocalElementLevelsConverter<ulong>(links, linkToItsFrequencyNumberConverter);
170  //                      var optimalVariantConverter = new OptimalVariantConverter<ulong>(links,
     ↪  sequenceToItsLocalElementLevelsConverter);
171  //
172  //                      var walker = new RightSequenceWalker<ulong>(links, new DefaultStack<ulong>(),
     ↪  (link) => constants.IsExternalReference(link) || links.IsPartialPoint(link));
173  //
174  //                      var unicodeSequencesOptions = new SequencesOptions<ulong>()
175  //                      {
176  //                          UseSequenceMarker = true,
177  //                          SequenceMarkerLink = unicodeSequenceMarker,
178  //                          UseIndex = true,
179  //                          Index = index,
180  //                          LinksToSequenceConverter = optimalVariantConverter,
181  //                          Walker = walker,
182  //                          UseGarbageCollection = true
183  //                      };
184  //
185  //                      var unicodeSequences = new Sequences(new SynchronizedLinks<ulong>(links),
     ↪  unicodeSequencesOptions);
186  //
187  //                      // Create some sequences
188  //                      var strings = _loremIpsumExample.Split(new[] { '\n', '\r' },
     ↪  StringSplitOptions.RemoveEmptyEntries);
189  //                      var arrays = strings.Select(x => x.Select(y =>
     ↪  addressToNumberConverter.Convert(y)).ToArray()).ToArray();
190  //                      for (int i = 0; i < arrays.Length; i++)
191  //                      {
192  //                          unicodeSequences.Create(arrays[i].ShiftRight());
193  //                      }
194  //
195  //                      var linksCountAfterCreation = links.Count();
196  //
197  //                      // get list of sequences links
198  //                      // for each sequence link
199  //                      //   create new sequence version
200  //                      //   if new sequence is not the same as sequence link
201  //                      //     delete sequence link
202  //                      //     collect garbadge
203  //                      unicodeSequences.CompactAll();
204  //
205  //                      var linksCountAfterCompactification = links.Count();
206  //
207  //                      Assert.True(linksCountAfterCompactification < linksCountAfterCreation);
208  //                  }
209  //          }
210  //      }
211  // }
```

```csharp
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 TLinkAddress = System.UInt64;
10
11  namespace Platform.Data.Doublets.Sequences.Tests
12  {
13      public class RationalNumbersTests
14      {
15          public ILinks<TLinkAddress> CreateLinks() => CreateLinks<TLinkAddress>(new
            ↪  IO.TemporaryFile());
16
17          public ILinks<TLinkAddress> CreateLinks<TLinkAddress>(string dataDbFilename)
18          {
19              var linksConstants = new
                ↪  LinksConstants<TLinkAddress>(enableExternalReferencesSupport: true);
20              return new UnitedMemoryLinks<TLinkAddress>(new
                ↪  FileMappedResizableDirectMemory(dataDbFilename),
                ↪  UnitedMemoryLinks<TLinkAddress>.DefaultLinksSizeStep, linksConstants,
                ↪  IndexTreeType.Default);
21          }
22
23          [Fact]
24          public void DecimalMinValueTest()
25          {
26              const decimal @decimal = decimal.MinValue;
27              var links = CreateLinks();
28              TLinkAddress negativeNumberMarker = links.Create();
29              AddressToRawNumberConverter<TLinkAddress> addressToRawNumberConverter = new();
30              RawNumberToAddressConverter<TLinkAddress> numberToAddressConverter = new();
31              BalancedVariantConverter<TLinkAddress> balancedVariantConverter = new(links);
32              BigIntegerToRawNumberSequenceConverter<TLinkAddress>
                ↪  bigIntegerToRawNumberSequenceConverter = new(links, addressToRawNumberConverter,
                ↪  balancedVariantConverter, negativeNumberMarker);
33              RawNumberSequenceToBigIntegerConverter<TLinkAddress>
                ↪  rawNumberSequenceToBigIntegerConverter = new(links, numberToAddressConverter,
                ↪  negativeNumberMarker);
34              DecimalToRationalConverter<TLinkAddress> decimalToRationalConverter = new(links,
                ↪  bigIntegerToRawNumberSequenceConverter);
35              RationalToDecimalConverter<TLinkAddress> rationalToDecimalConverter = new(links,
                ↪  rawNumberSequenceToBigIntegerConverter);
36              var rationalNumber = decimalToRationalConverter.Convert(@decimal);
37              var decimalFromRational = rationalToDecimalConverter.Convert(rationalNumber);
38              Assert.Equal(@decimal, decimalFromRational);
39          }
40
41          [Fact]
42          public void DecimalMaxValueTest()
43          {
44              const decimal @decimal = decimal.MaxValue;
45              var links = CreateLinks();
46              TLinkAddress negativeNumberMarker = links.Create();
47              AddressToRawNumberConverter<TLinkAddress> addressToRawNumberConverter = new();
48              RawNumberToAddressConverter<TLinkAddress> numberToAddressConverter = new();
49              BalancedVariantConverter<TLinkAddress> balancedVariantConverter = new(links);
50              BigIntegerToRawNumberSequenceConverter<TLinkAddress>
                ↪  bigIntegerToRawNumberSequenceConverter = new(links, addressToRawNumberConverter,
                ↪  balancedVariantConverter, negativeNumberMarker);
51              RawNumberSequenceToBigIntegerConverter<TLinkAddress>
                ↪  rawNumberSequenceToBigIntegerConverter = new(links, numberToAddressConverter,
                ↪  negativeNumberMarker);
52              DecimalToRationalConverter<TLinkAddress> decimalToRationalConverter = new(links,
                ↪  bigIntegerToRawNumberSequenceConverter);
53              RationalToDecimalConverter<TLinkAddress> rationalToDecimalConverter = new(links,
                ↪  rawNumberSequenceToBigIntegerConverter);
54              var rationalNumber = decimalToRationalConverter.Convert(@decimal);
55              var decimalFromRational = rationalToDecimalConverter.Convert(rationalNumber);
56              Assert.Equal(@decimal, decimalFromRational);
57          }
58
59          [Fact]
60          public void DecimalPositiveHalfTest()
```

```csharp
61          {
62              const decimal @decimal = 0.5M;
63              var links = CreateLinks();
64              TLinkAddress negativeNumberMarker = links.Create();
65              AddressToRawNumberConverter<TLinkAddress> addressToRawNumberConverter = new();
66              RawNumberToAddressConverter<TLinkAddress> numberToAddressConverter = new();
67              BalancedVariantConverter<TLinkAddress> balancedVariantConverter = new(links);
68              BigIntegerToRawNumberSequenceConverter<TLinkAddress>
                ↪  bigIntegerToRawNumberSequenceConverter = new(links, addressToRawNumberConverter,
                ↪  balancedVariantConverter, negativeNumberMarker);
69              RawNumberSequenceToBigIntegerConverter<TLinkAddress>
                ↪  rawNumberSequenceToBigIntegerConverter = new(links, numberToAddressConverter,
                ↪  negativeNumberMarker);
70              DecimalToRationalConverter<TLinkAddress> decimalToRationalConverter = new(links,
                ↪  bigIntegerToRawNumberSequenceConverter);
71              RationalToDecimalConverter<TLinkAddress> rationalToDecimalConverter = new(links,
                ↪  rawNumberSequenceToBigIntegerConverter);
72              var rationalNumber = decimalToRationalConverter.Convert(@decimal);
73              var decimalFromRational = rationalToDecimalConverter.Convert(rationalNumber);
74              Assert.Equal(@decimal, decimalFromRational);
75          }
76
77          [Fact]
78          public void DecimalNegativeHalfTest()
79          {
80              const decimal @decimal = -0.5M;
81              var links = CreateLinks();
82              TLinkAddress negativeNumberMarker = links.Create();
83              AddressToRawNumberConverter<TLinkAddress> addressToRawNumberConverter = new();
84              RawNumberToAddressConverter<TLinkAddress> numberToAddressConverter = new();
85              BalancedVariantConverter<TLinkAddress> balancedVariantConverter = new(links);
86              BigIntegerToRawNumberSequenceConverter<TLinkAddress>
                ↪  bigIntegerToRawNumberSequenceConverter = new(links, addressToRawNumberConverter,
                ↪  balancedVariantConverter, negativeNumberMarker);
87              RawNumberSequenceToBigIntegerConverter<TLinkAddress>
                ↪  rawNumberSequenceToBigIntegerConverter = new(links, numberToAddressConverter,
                ↪  negativeNumberMarker);
88              DecimalToRationalConverter<TLinkAddress> decimalToRationalConverter = new(links,
                ↪  bigIntegerToRawNumberSequenceConverter);
89              RationalToDecimalConverter<TLinkAddress> rationalToDecimalConverter = new(links,
                ↪  rawNumberSequenceToBigIntegerConverter);
90              var rationalNumber = decimalToRationalConverter.Convert(@decimal);
91              var decimalFromRational = rationalToDecimalConverter.Convert(rationalNumber);
92              Assert.Equal(@decimal, decimalFromRational);
93          }
94
95          [Fact]
96          public void DecimalOneTest()
97          {
98              const decimal @decimal = 1;
99              var links = CreateLinks();
100             TLinkAddress negativeNumberMarker = links.Create();
101             AddressToRawNumberConverter<TLinkAddress> addressToRawNumberConverter = new();
102             RawNumberToAddressConverter<TLinkAddress> numberToAddressConverter = new();
103             BalancedVariantConverter<TLinkAddress> balancedVariantConverter = new(links);
104             BigIntegerToRawNumberSequenceConverter<TLinkAddress>
                ↪  bigIntegerToRawNumberSequenceConverter = new(links, addressToRawNumberConverter,
                ↪  balancedVariantConverter, negativeNumberMarker);
105             RawNumberSequenceToBigIntegerConverter<TLinkAddress>
                ↪  rawNumberSequenceToBigIntegerConverter = new(links, numberToAddressConverter,
                ↪  negativeNumberMarker);
106             DecimalToRationalConverter<TLinkAddress> decimalToRationalConverter = new(links,
                ↪  bigIntegerToRawNumberSequenceConverter);
107             RationalToDecimalConverter<TLinkAddress> rationalToDecimalConverter = new(links,
                ↪  rawNumberSequenceToBigIntegerConverter);
108             var rationalNumber = decimalToRationalConverter.Convert(@decimal);
109             var decimalFromRational = rationalToDecimalConverter.Convert(rationalNumber);
110             Assert.Equal(@decimal, decimalFromRational);
111         }
112
113         [Fact]
114         public void DecimalMinusOneTest()
115         {
116             const decimal @decimal = -1;
117             var links = CreateLinks();
118             TLinkAddress negativeNumberMarker = links.Create();
119             AddressToRawNumberConverter<TLinkAddress> addressToRawNumberConverter = new();
```

```
120        RawNumberToAddressConverter<TLinkAddress> numberToAddressConverter = new();
121        BalancedVariantConverter<TLinkAddress> balancedVariantConverter = new(links);
122        BigIntegerToRawNumberSequenceConverter<TLinkAddress>
      ↪    bigIntegerToRawNumberSequenceConverter = new(links, addressToRawNumberConverter,
      ↪    balancedVariantConverter, negativeNumberMarker);
123        RawNumberSequenceToBigIntegerConverter<TLinkAddress>
      ↪    rawNumberSequenceToBigIntegerConverter = new(links, numberToAddressConverter,
      ↪    negativeNumberMarker);
124        DecimalToRationalConverter<TLinkAddress> decimalToRationalConverter = new(links,
      ↪    bigIntegerToRawNumberSequenceConverter);
125        RationalToDecimalConverter<TLinkAddress> rationalToDecimalConverter = new(links,
      ↪    rawNumberSequenceToBigIntegerConverter);
126        var rationalNumber = decimalToRationalConverter.Convert(@decimal);
127        var decimalFromRational = rationalToDecimalConverter.Convert(rationalNumber);
128        Assert.Equal(@decimal, decimalFromRational);
129      }
130    }
131  }
```

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

```
1   // using System;
2   // using System.Collections.Generic;
3   // using System.Diagnostics;
4   // using System.Linq;
5   // using Xunit;
6   // using Platform.Data.Sequences;
7   // using Platform.Data.Doublets.Sequences.Converters;
8   // using Platform.Data.Doublets.Sequences.Walkers;
9   // using Platform.Data.Doublets.Sequences;
10  //
11  // namespace Platform.Data.Doublets.Sequences.Tests
12  // {
13  //     public static class ReadSequenceTests
14  //     {
15  //         [Fact]
16  //         public static void ReadSequenceTest()
17  //         {
18  //             const long sequenceLength = 2000;
19  //
20  //             using (var scope = new TempLinksTestScope(useSequences: false))
21  //             {
22  //                 var links = scope.Links;
23  //                 var sequences = new Sequences(links, new SequencesOptions<ulong> { Walker =
     ↪  new LeveledSequenceWalker<ulong>(links) });
24  //
25  //                 var sequence = new ulong[sequenceLength];
26  //                 for (var i = 0; i < sequenceLength; i++)
27  //                 {
28  //                     sequence[i] = links.Create();
29  //                 }
30  //
31  //                 var balancedVariantConverter = new BalancedVariantConverter<ulong>(links);
32  //
33  //                 var sw1 = Stopwatch.StartNew();
34  //                 var balancedVariant = balancedVariantConverter.Convert(sequence); sw1.Stop();
35  //
36  //                 var sw2 = Stopwatch.StartNew();
37  //                 var readSequence1 = sequences.ToList(balancedVariant); sw2.Stop();
38  //
39  //                 var sw3 = Stopwatch.StartNew();
40  //                 var readSequence2 = new List<ulong>();
41  //                 SequenceWalker.WalkRight(balancedVariant,
42  //                                          links.GetSource,
43  //                                          links.GetTarget,
44  //                                          links.IsPartialPoint,
45  //                                          readSequence2.Add);
46  //                 sw3.Stop();
47  //
48  //                 Assert.True(sequence.SequenceEqual(readSequence1));
49  //
50  //                 Assert.True(sequence.SequenceEqual(readSequence2));
51  //
52  //                 // Assert.True(sw2.Elapsed < sw3.Elapsed);
53  //
54  //                 Console.WriteLine($"Stack-based walker: {sw3.Elapsed}, Level-based reader:
     ↪  {sw2.Elapsed}");
55  //
```

```
56   //                    for (var i = 0; i < sequenceLength; i++)
57   //                    {
58   //                        links.Delete(sequence[i]);
59   //                    }
60   //                }
61   //            }
62   //        }
63   // }
```

1.68 ./csharp/Platform.Data.Doublets.Sequences.Tests/SequencesTests.cs

```
1    // using System;
2    // using System.Collections.Generic;
3    // using System.Diagnostics;
4    // using System.Linq;
5    // using Xunit;
6    // using Platform.Collections;
7    // using Platform.Collections.Arrays;
8    // using Platform.Random;
9    // using Platform.IO;
10   // using Platform.Singletons;
11   // using Platform.Data.Doublets.Sequences;
12   // using Platform.Data.Doublets.Sequences.Frequencies.Cache;
13   // using Platform.Data.Doublets.Sequences.Frequencies.Counters;
14   // using Platform.Data.Doublets.Sequences.Converters;
15   // using Platform.Data.Doublets.Unicode;
16   //
17   // namespace Platform.Data.Doublets.Sequences.Tests
18   // {
19   //     public static class SequencesTests
20   //     {
21   //         private static readonly LinksConstants<ulong> _constants =
     ↪   Default<LinksConstants<ulong>>.Instance;
22   //
23   //         static SequencesTests()
24   //         {
25   //             // Trigger static constructor to not mess with perfomance measurements
26   //             _ = BitString.GetBitMaskFromIndex(1);
27   //         }
28   //
29   //         [Fact]
30   //         public static void CreateAllVariantsTest()
31   //         {
32   //             const long sequenceLength = 8;
33   //
34   //             using (var scope = new TempLinksTestScope(useSequences: true))
35   //             {
36   //                 var links = scope.Links;
37   //                 var sequences = scope.Sequences;
38   //
39   //                 var sequence = new ulong[sequenceLength];
40   //                 for (var i = 0; i < sequenceLength; i++)
41   //                 {
42   //                     sequence[i] = links.Create();
43   //                 }
44   //
45   //                 var sw1 = Stopwatch.StartNew();
46   //                 var results1 = sequences.CreateAllVariants1(sequence); sw1.Stop();
47   //
48   //                 var sw2 = Stopwatch.StartNew();
49   //                 var results2 = sequences.CreateAllVariants2(sequence); sw2.Stop();
50   //
51   //                 Assert.True(results1.Count > results2.Length);
52   //                 Assert.True(sw1.Elapsed > sw2.Elapsed);
53   //
54   //                 for (var i = 0; i < sequenceLength; i++)
55   //                 {
56   //                     links.Delete(sequence[i]);
57   //                 }
58   //
59   //                 Assert.True(links.Count() == 0);
60   //             }
61   //         }
62   //
63   //         //[Fact]
64   //         //public void CUDTest()
65   //         //{
66   //         //     var tempFilename = Path.GetTempFileName();
67   //
```

```
68  //              //      const long sequenceLength = 8;
69  //
70  //              //      const ulong itself = LinksConstants.Itself;
71  //
72  //              //      using (var memoryAdapter = new ResizableDirectMemoryLinks(tempFilename,
    ↪  DefaultLinksSizeStep))
73  //              //      using (var links = new Links(memoryAdapter))
74  //              //      {
75  //              //          var sequence = new ulong[sequenceLength];
76  //              //          for (var i = 0; i < sequenceLength; i++)
77  //              //              sequence[i] = links.Create(itself, itself);
78  //
79  //              //          SequencesOptions o = new SequencesOptions();
80  //
81  //              // TODO: Из числа в bool значения o.UseSequenceMarker = ((value & 1) != 0)
82  //              //          o.
83  //
84  //              //          var sequences = new Sequences(links);
85  //
86  //              //          var sw1 = Stopwatch.StartNew();
87  //              //          var results1 = sequences.CreateAllVariants1(sequence); sw1.Stop();
88  //
89  //              //          var sw2 = Stopwatch.StartNew();
90  //              //          var results2 = sequences.CreateAllVariants2(sequence); sw2.Stop();
91  //
92  //              //          Assert.True(results1.Count > results2.Length);
93  //              //          Assert.True(sw1.Elapsed > sw2.Elapsed);
94  //
95  //              //          for (var i = 0; i < sequenceLength; i++)
96  //              //              links.Delete(sequence[i]);
97  //              //      }
98  //
99  //              //      File.Delete(tempFilename);
100 //              //}
101 //
102 //              [Fact]
103 //              public static void AllVariantsSearchTest()
104 //              {
105 //                  const long sequenceLength = 8;
106 //
107 //                  using (var scope = new TempLinksTestScope(useSequences: true))
108 //                  {
109 //                      var links = scope.Links;
110 //                      var sequences = scope.Sequences;
111 //
112 //                      var sequence = new ulong[sequenceLength];
113 //                      for (var i = 0; i < sequenceLength; i++)
114 //                      {
115 //                          sequence[i] = links.Create();
116 //                      }
117 //
118 //                      var createResults =
    ↪  sequences.CreateAllVariants2(sequence).Distinct().ToArray();
119 //
120 //                      //for (int i = 0; i < createResults.Length; i++)
121 //                      //    sequences.Create(createResults[i]);
122 //
123 //                      var sw0 = Stopwatch.StartNew();
124 //                      var searchResults0 = sequences.GetAllMatchingSequences0(sequence); sw0.Stop();
125 //
126 //                      var sw1 = Stopwatch.StartNew();
127 //                      var searchResults1 = sequences.GetAllMatchingSequences1(sequence); sw1.Stop();
128 //
129 //                      var sw2 = Stopwatch.StartNew();
130 //                      var searchResults2 = sequences.Each1(sequence); sw2.Stop();
131 //
132 //                      var sw3 = Stopwatch.StartNew();
133 //                      var searchResults3 = sequences.Each(sequence.ShiftRight()); sw3.Stop();
134 //
135 //                      var intersection0 = createResults.Intersect(searchResults0).ToList();
136 //                      Assert.True(intersection0.Count == searchResults0.Count);
137 //                      Assert.True(intersection0.Count == createResults.Length);
138 //
139 //                      var intersection1 = createResults.Intersect(searchResults1).ToList();
140 //                      Assert.True(intersection1.Count == searchResults1.Count);
141 //                      Assert.True(intersection1.Count == createResults.Length);
142 //
```

```
143  //                    var intersection2 = createResults.Intersect(searchResults2).ToList();
144  //                    Assert.True(intersection2.Count == searchResults2.Count);
145  //                    Assert.True(intersection2.Count == createResults.Length);
146  //
147  //                    var intersection3 = createResults.Intersect(searchResults3).ToList();
148  //                    Assert.True(intersection3.Count == searchResults3.Count);
149  //                    Assert.True(intersection3.Count == createResults.Length);
150  //
151  //                    for (var i = 0; i < sequenceLength; i++)
152  //                    {
153  //                        links.Delete(sequence[i]);
154  //                    }
155  //                }
156  //            }
157  //
158  //            [Fact]
159  //            public static void BalancedVariantSearchTest()
160  //            {
161  //                const long sequenceLength = 200;
162  //
163  //                using (var scope = new TempLinksTestScope(useSequences: true))
164  //                {
165  //                    var links = scope.Links;
166  //                    var sequences = scope.Sequences;
167  //
168  //                    var sequence = new ulong[sequenceLength];
169  //                    for (var i = 0; i < sequenceLength; i++)
170  //                    {
171  //                        sequence[i] = links.Create();
172  //                    }
173  //
174  //                    var balancedVariantConverter = new BalancedVariantConverter<ulong>(links);
175  //
176  //                    var sw1 = Stopwatch.StartNew();
177  //                    var balancedVariant = balancedVariantConverter.Convert(sequence); sw1.Stop();
178  //
179  //                    var sw2 = Stopwatch.StartNew();
180  //                    var searchResults2 = sequences.GetAllMatchingSequences0(sequence); sw2.Stop();
181  //
182  //                    var sw3 = Stopwatch.StartNew();
183  //                    var searchResults3 = sequences.GetAllMatchingSequences1(sequence); sw3.Stop();
184  //
185  //                    // На количестве в 200 элементов это будет занимать вечность
186  //                    //var sw4 = Stopwatch.StartNew();
187  //                    //var searchResults4 = sequences.Each(sequence); sw4.Stop();
188  //
189  //                    Assert.True(searchResults2.Count == 1 && balancedVariant ==
     ↪   searchResults2[0]);
190  //
191  //                    Assert.True(searchResults3.Count == 1 && balancedVariant ==
     ↪   searchResults3.First());
192  //
193  //                    //Assert.True(sw1.Elapsed < sw2.Elapsed);
194  //
195  //                    for (var i = 0; i < sequenceLength; i++)
196  //                    {
197  //                        links.Delete(sequence[i]);
198  //                    }
199  //                }
200  //            }
201  //
202  //            [Fact]
203  //            public static void AllPartialVariantsSearchTest()
204  //            {
205  //                const long sequenceLength = 8;
206  //
207  //                using (var scope = new TempLinksTestScope(useSequences: true))
208  //                {
209  //                    var links = scope.Links;
210  //                    var sequences = scope.Sequences;
211  //
212  //                    var sequence = new ulong[sequenceLength];
213  //                    for (var i = 0; i < sequenceLength; i++)
214  //                    {
215  //                        sequence[i] = links.Create();
216  //                    }
217  //
```

```
218  //                        var createResults = sequences.CreateAllVariants2(sequence);
219  //
220  //                        //var createResultsStrings = createResults.Select(x => x + ": " +
     ↪  sequences.FormatSequence(x)).ToList();
221  //                        //Global.Trash = createResultsStrings;
222  //
223  //                        var partialSequence = new ulong[sequenceLength - 2];
224  //
225  //                        Array.Copy(sequence, 1, partialSequence, 0, (int)sequenceLength - 2);
226  //
227  //                        var sw1 = Stopwatch.StartNew();
228  //                        var searchResults1 =
     ↪  sequences.GetAllPartiallyMatchingSequences0(partialSequence); sw1.Stop();
229  //
230  //                        var sw2 = Stopwatch.StartNew();
231  //                        var searchResults2 =
     ↪  sequences.GetAllPartiallyMatchingSequences1(partialSequence); sw2.Stop();
232  //
233  //                        //var sw3 = Stopwatch.StartNew();
234  //                        //var searchResults3 =
     ↪  sequences.GetAllPartiallyMatchingSequences2(partialSequence); sw3.Stop();
235  //
236  //                        var sw4 = Stopwatch.StartNew();
237  //                        var searchResults4 =
     ↪  sequences.GetAllPartiallyMatchingSequences3(partialSequence); sw4.Stop();
238  //
239  //                        //Global.Trash = searchResults3;
240  //
241  //                        //var searchResults1Strings = searchResults1.Select(x => x + ": " +
     ↪  sequences.FormatSequence(x)).ToList();
242  //                        //Global.Trash = searchResults1Strings;
243  //
244  //                        var intersection1 = createResults.Intersect(searchResults1).ToList();
245  //                        Assert.True(intersection1.Count == createResults.Length);
246  //
247  //                        var intersection2 = createResults.Intersect(searchResults2).ToList();
248  //                        Assert.True(intersection2.Count == createResults.Length);
249  //
250  //                        var intersection4 = createResults.Intersect(searchResults4).ToList();
251  //                        Assert.True(intersection4.Count == createResults.Length);
252  //
253  //                        for (var i = 0; i < sequenceLength; i++)
254  //                        {
255  //                            links.Delete(sequence[i]);
256  //                        }
257  //                    }
258  //                }
259  //
260  //            [Fact]
261  //            public static void BalancedPartialVariantsSearchTest()
262  //            {
263  //                const long sequenceLength = 200;
264  //
265  //                using (var scope = new TempLinksTestScope(useSequences: true))
266  //                {
267  //                    var links = scope.Links;
268  //                    var sequences = scope.Sequences;
269  //
270  //                    var sequence = new ulong[sequenceLength];
271  //                    for (var i = 0; i < sequenceLength; i++)
272  //                    {
273  //                        sequence[i] = links.Create();
274  //                    }
275  //
276  //                    var balancedVariantConverter = new BalancedVariantConverter<ulong>(links);
277  //
278  //                    var balancedVariant = balancedVariantConverter.Convert(sequence);
279  //
280  //                    var partialSequence = new ulong[sequenceLength - 2];
281  //
282  //                    Array.Copy(sequence, 1, partialSequence, 0, (int)sequenceLength - 2);
283  //
284  //                    var sw1 = Stopwatch.StartNew();
285  //                    var searchResults1 =
     ↪  sequences.GetAllPartiallyMatchingSequences0(partialSequence); sw1.Stop();
286  //
287  //                    var sw2 = Stopwatch.StartNew();
```

```
288  //                    var searchResults2 =
  ↪ sequences.GetAllPartiallyMatchingSequences1(partialSequence); sw2.Stop();
289  //
290  //                    Assert.True(searchResults1.Count == 1 && balancedVariant ==
  ↪ searchResults1[0]);
291  //
292  //                    Assert.True(searchResults2.Count == 1 && balancedVariant ==
  ↪ searchResults2.First());
293  //
294  //                    for (var i = 0; i < sequenceLength; i++)
295  //                    {
296  //                        links.Delete(sequence[i]);
297  //                    }
298  //                }
299  //            }
300  //
301  //            [Fact(Skip = "Correct implementation is pending")]
302  //            public static void PatternMatchTest()
303  //            {
304  //                var zeroOrMany = Sequences.ZeroOrMany;
305  //
306  //                using (var scope = new TempLinksTestScope(useSequences: true))
307  //                {
308  //                    var links = scope.Links;
309  //                    var sequences = scope.Sequences;
310  //
311  //                    var e1 = links.Create();
312  //                    var e2 = links.Create();
313  //
314  //                    var sequence = new[]
315  //                    {
316  //                        e1, e2, e1, e2 // mama / papa
317  //                    };
318  //
319  //                    var balancedVariantConverter = new BalancedVariantConverter<ulong>(links);
320  //
321  //                    var balancedVariant = balancedVariantConverter.Convert(sequence);
322  //
323  //                    // 1: [1]
324  //                    // 2: [2]
325  //                    // 3: [1,2]
326  //                    // 4: [1,2,1,2]
327  //
328  //                    var doublet = links.GetSource(balancedVariant);
329  //
330  //                    var matchedSequences1 = sequences.MatchPattern(e2, e1, zeroOrMany);
331  //
332  //                    Assert.True(matchedSequences1.Count == 0);
333  //
334  //                    var matchedSequences2 = sequences.MatchPattern(zeroOrMany, e2, e1);
335  //
336  //                    Assert.True(matchedSequences2.Count == 0);
337  //
338  //                    var matchedSequences3 = sequences.MatchPattern(e1, zeroOrMany, e1);
339  //
340  //                    Assert.True(matchedSequences3.Count == 0);
341  //
342  //                    var matchedSequences4 = sequences.MatchPattern(e1, zeroOrMany, e2);
343  //
344  //                    Assert.Contains(doublet, matchedSequences4);
345  //                    Assert.Contains(balancedVariant, matchedSequences4);
346  //
347  //                    for (var i = 0; i < sequence.Length; i++)
348  //                    {
349  //                        links.Delete(sequence[i]);
350  //                    }
351  //                }
352  //            }
353  //
354  //            [Fact]
355  //            public static void IndexTest()
356  //            {
357  //                using (var scope = new TempLinksTestScope(new SequencesOptions<ulong> { UseIndex
  ↪ = true }, useSequences: true))
358  //                {
359  //                    var links = scope.Links;
360  //                    var sequences = scope.Sequences;
```

```
361   //                    var index = sequences.Options.Index;
362   //
363   //                    var e1 = links.Create();
364   //                    var e2 = links.Create();
365   //
366   //                    var sequence = new[]
367   //                    {
368   //                        e1, e2, e1, e2 // mama / papa
369   //                    };
370   //
371   //                    Assert.False(index.MightContain(sequence));
372   //
373   //                    index.Add(sequence);
374   //
375   //                    Assert.True(index.MightContain(sequence));
376   //                }
377   //            }
378   //            private static readonly string _exampleText =
379   //                @"([english
      ↪  version](https://github.com/Konard/LinksPlatform/wiki/About-the-beginning))
380   //
381   // 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?
382   //
383   // [![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)
384   //
385   // 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?
386   //
387   // [![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)
388   //
389   // А что если повторить? Сделать копию? Создать дубликат? Из одного сделать два? Может это быть
      ↪  так? Инверсия? Отражение? Сумма?
390   //
391   // [![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)
392   //
393   // 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?
394   //
395   // [![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)
396   //
397   // 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?
398   //
399   // [![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)
400   //
401   // 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?
402   //
403   // [![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)
```

```
404      //
405      // 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?
406      //
407      // [![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)
408      //
409      // 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?
410      //
411      // [![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)
412      //
413      // 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?
414      //
415      // [![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)
416      //
417      // 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?
418      //
419      // [![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.c
    ↪    om/Konard/LinksPlatform/master/doc/Intro/10.png)
420      //
421      // На один уровень внутрь (вниз)? Или на один уровень во вне (вверх)? Или это можно назвать
    ↪    шагом рекурсии или фрактала?
422      //
423      // [![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.githubuserc
    ↪    ontent.com/Konard/LinksPlatform/master/doc/Intro/11.png)
424      //
425      // Последовательность? Массив? Список? Множество? Объект? Таблица? Элементы? Цвета? Символы?
    ↪    Буквы? Слово? Цифры? Число? Алфавит? Дерево? Сеть? Граф? Гиперграф?
426      //
427      // [![белая обычная и направленная связи со структурой из 8 цветных элементов
    ↪    последовательности, чёрная типизированная связь со структурой из 8 цветных элементов последо
    ↪    вательности](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/Int
    ↪    ro/12.png)
428      //
429      // ...
430      //
431      // [![анимация](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/intro-an
    ↪    imation-500.gif
    ↪    ""анимация"")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/intro
    ↪    -animation-500.gif)";
432      //         private static readonly string _exampleLoremIpsumText =
433      //             @"Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor
    ↪    incididunt ut labore et dolore magna aliqua.
434      // Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea
    ↪    commodo consequat.";
435      //
436      //         [Fact]
437      //         public static void CompressionTest()
```

```
438  //            {
439  //                using (var scope = new TempLinksTestScope(useSequences: true))
440  //                {
441  //                    var links = scope.Links;
442  //                    var sequences = scope.Sequences;
443  //
444  //                    var e1 = links.Create();
445  //                    var e2 = links.Create();
446  //
447  //                    var sequence = new[]
448  //                    {
449  //                        e1, e2, e1, e2 // mama / papa / template [(m/p), a] { [1] [2] [1] [2] }
450  //                    };
451  //
452  //                    var balancedVariantConverter = new
     //  BalancedVariantConverter<ulong>(links.Unsync);
453  //                    var totalSequenceSymbolFrequencyCounter = new
     //  TotalSequenceSymbolFrequencyCounter<ulong>(links.Unsync);
454  //                    var doubletFrequenciesCache = new LinkFrequenciesCache<ulong>(links.Unsync,
     //  totalSequenceSymbolFrequencyCounter);
455  //                    var compressingConverter = new CompressingConverter<ulong>(links.Unsync,
     //  balancedVariantConverter, doubletFrequenciesCache);
456  //
457  //                    var compressedVariant = compressingConverter.Convert(sequence);
458  //
459  //                    // 1: [1]       (1->1) point
460  //                    // 2: [2]       (2->2) point
461  //                    // 3: [1,2]     (1->2) doublet
462  //                    // 4: [1,2,1,2] (3->3) doublet
463  //
464  //                    Assert.True(links.GetSource(links.GetSource(compressedVariant)) ==
     //  sequence[0]);
465  //                    Assert.True(links.GetTarget(links.GetSource(compressedVariant)) ==
     //  sequence[1]);
466  //                    Assert.True(links.GetSource(links.GetTarget(compressedVariant)) ==
     //  sequence[2]);
467  //                    Assert.True(links.GetTarget(links.GetTarget(compressedVariant)) ==
     //  sequence[3]);
468  //
469  //                    var source = _constants.SourcePart;
470  //                    var target = _constants.TargetPart;
471  //
472  //                    Assert.True(links.GetByKeys(compressedVariant, source, source) ==
     //  sequence[0]);
473  //                    Assert.True(links.GetByKeys(compressedVariant, source, target) ==
     //  sequence[1]);
474  //                    Assert.True(links.GetByKeys(compressedVariant, target, source) ==
     //  sequence[2]);
475  //                    Assert.True(links.GetByKeys(compressedVariant, target, target) ==
     //  sequence[3]);
476  //
477  //                    // 4 - length of sequence
478  //                    Assert.True(links.GetSquareMatrixSequenceElementByIndex(compressedVariant, 4,
     //  0) == sequence[0]);
479  //                    Assert.True(links.GetSquareMatrixSequenceElementByIndex(compressedVariant, 4,
     //  1) == sequence[1]);
480  //                    Assert.True(links.GetSquareMatrixSequenceElementByIndex(compressedVariant, 4,
     //  2) == sequence[2]);
481  //                    Assert.True(links.GetSquareMatrixSequenceElementByIndex(compressedVariant, 4,
     //  3) == sequence[3]);
482  //                }
483  //            }
484  //
485  //            [Fact]
486  //            public static void CompressionEfficiencyTest()
487  //            {
488  //                var strings = _exampleLoremIpsumText.Split(new[] { '\n', '\r' },
     //  StringSplitOptions.RemoveEmptyEntries);
489  //                var arrays = strings.Select(UnicodeMap.FromStringToLinkArray).ToArray();
490  //                var totalCharacters = arrays.Select(x => x.Length).Sum();
491  //
492  //                using (var scope1 = new TempLinksTestScope(useSequences: true))
493  //                using (var scope2 = new TempLinksTestScope(useSequences: true))
494  //                using (var scope3 = new TempLinksTestScope(useSequences: true))
495  //                {
496  //                    scope1.Links.Unsync.UseUnicode();
```

```
497  //                    scope2.Links.Unsync.UseUnicode();
498  //                    scope3.Links.Unsync.UseUnicode();
499  //
500  //                    var balancedVariantConverter1 = new
     BalancedVariantConverter<ulong>(scope1.Links.Unsync);
501  //                    var totalSequenceSymbolFrequencyCounter = new
     TotalSequenceSymbolFrequencyCounter<ulong>(scope1.Links.Unsync);
502  //                    var linkFrequenciesCache1 = new
     LinkFrequenciesCache<ulong>(scope1.Links.Unsync, totalSequenceSymbolFrequencyCounter);
503  //                    var compressor1 = new CompressingConverter<ulong>(scope1.Links.Unsync,
     balancedVariantConverter1, linkFrequenciesCache1, doInitialFrequenciesIncrement: false);
504  //
505  //                    //var compressor2 = scope2.Sequences;
506  //                    var compressor3 = scope3.Sequences;
507  //
508  //                    var constants = Default<LinksConstants<ulong>>.Instance;
509  //
510  //                    var sequences = compressor3;
511  //                    //var meaningRoot = links.CreatePoint();
512  //                    //var unaryOne = links.CreateAndUpdate(meaningRoot, constants.Itself);
513  //                    //var frequencyMarker = links.CreateAndUpdate(meaningRoot, constants.Itself);
514  //                    //var frequencyPropertyMarker = links.CreateAndUpdate(meaningRoot,
     constants.Itself);
515  //
516  //                    //var unaryNumberToAddressConverter = new
     UnaryNumberToAddressAddOperationConverter<ulong>(links, unaryOne);
517  //                    //var unaryNumberIncrementer = new UnaryNumberIncrementer<ulong>(links,
     unaryOne);
518  //                    //var frequencyIncrementer = new FrequencyIncrementer<ulong>(links,
     frequencyMarker, unaryOne, unaryNumberIncrementer);
519  //                    //var frequencyPropertyOperator = new FrequencyPropertyOperator<ulong>(links,
     frequencyPropertyMarker, frequencyMarker);
520  //                    //var linkFrequencyIncrementer = new LinkFrequencyIncrementer<ulong>(links,
     frequencyPropertyOperator, frequencyIncrementer);
521  //                    //var linkToItsFrequencyNumberConverter = new
     LinkToItsFrequencyNumberConveter<ulong>(links, frequencyPropertyOperator,
     unaryNumberToAddressConverter);
522  //
523  //                    var linkFrequenciesCache3 = new
     LinkFrequenciesCache<ulong>(scope3.Links.Unsync, totalSequenceSymbolFrequencyCounter);
524  //
525  //                    var linkToItsFrequencyNumberConverter = new
     FrequenciesCacheBasedLinkToItsFrequencyNumberConverter<ulong>(linkFrequenciesCache3);
526  //
527  //                    var sequenceToItsLocalElementLevelsConverter = new
     SequenceToItsLocalElementLevelsConverter<ulong>(scope3.Links.Unsync,
     linkToItsFrequencyNumberConverter);
528  //                    var optimalVariantConverter = new
     OptimalVariantConverter<ulong>(scope3.Links.Unsync,
     sequenceToItsLocalElementLevelsConverter);
529  //
530  //                    var compressed1 = new ulong[arrays.Length];
531  //                    var compressed2 = new ulong[arrays.Length];
532  //                    var compressed3 = new ulong[arrays.Length];
533  //
534  //                    var START = 0;
535  //                    var END = arrays.Length;
536  //
537  //                    //for (int i = START; i < END; i++)
538  //                    //    linkFrequenciesCache1.IncrementFrequencies(arrays[i]);
539  //
540  //                    var initialCount1 = scope2.Links.Unsync.Count();
541  //
542  //                    var sw1 = Stopwatch.StartNew();
543  //
544  //                    for (int i = START; i < END; i++)
545  //                    {
546  //                        linkFrequenciesCache1.IncrementFrequencies(arrays[i]);
547  //                        compressed1[i] = compressor1.Convert(arrays[i]);
548  //                    }
549  //
550  //                    var elapsed1 = sw1.Elapsed;
551  //
552  //                    var balancedVariantConverter2 = new
     BalancedVariantConverter<ulong>(scope2.Links.Unsync);
553  //
```

```
554  //                        var initialCount2 = scope2.Links.Unsync.Count();
555  //
556  //                        var sw2 = Stopwatch.StartNew();
557  //
558  //                        for (int i = START; i < END; i++)
559  //                        {
560  //                            compressed2[i] = balancedVariantConverter2.Convert(arrays[i]);
561  //                        }
562  //
563  //                        var elapsed2 = sw2.Elapsed;
564  //
565  //                        for (int i = START; i < END; i++)
566  //                        {
567  //                            linkFrequenciesCache3.IncrementFrequencies(arrays[i]);
568  //                        }
569  //
570  //                        var initialCount3 = scope3.Links.Unsync.Count();
571  //
572  //                        var sw3 = Stopwatch.StartNew();
573  //
574  //                        for (int i = START; i < END; i++)
575  //                        {
576  //                            //linkFrequenciesCache3.IncrementFrequencies(arrays[i]);
577  //                            compressed3[i] = optimalVariantConverter.Convert(arrays[i]);
578  //                        }
579  //
580  //                        var elapsed3 = sw3.Elapsed;
581  //
582  //                        Console.WriteLine($"Compressor: {elapsed1}, Balanced variant: {elapsed2},
     ↪  Optimal variant: {elapsed3}");
583  //
584  //                        // Assert.True(elapsed1 > elapsed2);
585  //
586  //                        // Checks
587  //                        for (int i = START; i < END; i++)
588  //                        {
589  //                            var sequence1 = compressed1[i];
590  //                            var sequence2 = compressed2[i];
591  //                            var sequence3 = compressed3[i];
592  //
593  //                            var decompress1 = UnicodeMap.FromSequenceLinkToString(sequence1,
     ↪  scope1.Links.Unsync);
594  //
595  //                            var decompress2 = UnicodeMap.FromSequenceLinkToString(sequence2,
     ↪  scope2.Links.Unsync);
596  //
597  //                            var decompress3 = UnicodeMap.FromSequenceLinkToString(sequence3,
     ↪  scope3.Links.Unsync);
598  //
599  //                            var structure1 = scope1.Links.Unsync.FormatStructure(sequence1, link =>
     ↪  link.IsPartialPoint());
600  //                            var structure2 = scope2.Links.Unsync.FormatStructure(sequence2, link =>
     ↪  link.IsPartialPoint());
601  //                            var structure3 = scope3.Links.Unsync.FormatStructure(sequence3, link =>
     ↪  link.IsPartialPoint());
602  //
603  //                            //if (sequence1 != Constants.Null && sequence2 != Constants.Null &&
     ↪  arrays[i].Length > 3)
604  //                            //    Assert.False(structure1 == structure2);
605  //                            //if (sequence3 != Constants.Null && sequence2 != Constants.Null &&
     ↪  arrays[i].Length > 3)
606  //                            //    Assert.False(structure3 == structure2);
607  //
608  //                            Assert.True(strings[i] == decompress1 && decompress1 == decompress2);
609  //                            Assert.True(strings[i] == decompress3 && decompress3 == decompress2);
610  //                        }
611  //
612  //                        Assert.True((int)(scope1.Links.Unsync.Count() - initialCount1) <
     ↪  totalCharacters);
613  //                        Assert.True((int)(scope2.Links.Unsync.Count() - initialCount2) <
     ↪  totalCharacters);
614  //                        Assert.True((int)(scope3.Links.Unsync.Count() - initialCount3) <
     ↪  totalCharacters);
615  //
```

```
616  //                    Console.WriteLine($"{(double)(scope1.Links.Unsync.Count() - initialCount1) /
     ↪  totalCharacters} | {(double)(scope2.Links.Unsync.Count() - initialCount2) / totalCharacters}
     ↪  | {(double)(scope3.Links.Unsync.Count() - initialCount3) / totalCharacters}");
617  //
618  //                    Assert.True(scope1.Links.Unsync.Count() - initialCount1 <
     ↪  scope2.Links.Unsync.Count() - initialCount2);
619  //                    Assert.True(scope3.Links.Unsync.Count() - initialCount3 <
     ↪  scope2.Links.Unsync.Count() - initialCount2);
620  //
621  //                    var duplicateProvider1 = new
     ↪  DuplicateSegmentsProvider<ulong>(scope1.Links.Unsync, scope1.Sequences);
622  //                    var duplicateProvider2 = new
     ↪  DuplicateSegmentsProvider<ulong>(scope2.Links.Unsync, scope2.Sequences);
623  //                    var duplicateProvider3 = new
     ↪  DuplicateSegmentsProvider<ulong>(scope3.Links.Unsync, scope3.Sequences);
624  //
625  //                    var duplicateCounter1 = new
     ↪  DuplicateSegmentsCounter<ulong>(duplicateProvider1);
626  //                    var duplicateCounter2 = new
     ↪  DuplicateSegmentsCounter<ulong>(duplicateProvider2);
627  //                    var duplicateCounter3 = new
     ↪  DuplicateSegmentsCounter<ulong>(duplicateProvider3);
628  //
629  //                    var duplicates1 = duplicateCounter1.Count();
630  //
631  //                    ConsoleHelpers.Debug("------");
632  //
633  //                    var duplicates2 = duplicateCounter2.Count();
634  //
635  //                    ConsoleHelpers.Debug("------");
636  //
637  //                    var duplicates3 = duplicateCounter3.Count();
638  //
639  //                    Console.WriteLine($"{duplicates1} | {duplicates2} | {duplicates3}");
640  //
641  //                    linkFrequenciesCache1.ValidateFrequencies();
642  //                    linkFrequenciesCache3.ValidateFrequencies();
643  //                }
644  //            }
645  //
646  //            [Fact]
647  //            public static void CompressionStabilityTest()
648  //            {
649  //                // TODO: Fix bug (do a separate test)
650  //                //const ulong minNumbers = 0;
651  //                //const ulong maxNumbers = 1000;
652  //
653  //                const ulong minNumbers = 10000;
654  //                const ulong maxNumbers = 12500;
655  //
656  //                var strings = new List<string>();
657  //
658  //                for (ulong i = minNumbers; i < maxNumbers; i++)
659  //                {
660  //                    strings.Add(i.ToString());
661  //                }
662  //
663  //                var arrays = strings.Select(UnicodeMap.FromStringToLinkArray).ToArray();
664  //                var totalCharacters = arrays.Select(x => x.Length).Sum();
665  //
666  //                using (var scope1 = new TempLinksTestScope(useSequences: true, sequencesOptions:
     ↪  new SequencesOptions<ulong> { UseCompression = true,
     ↪  EnforceSingleSequenceVersionOnWriteBasedOnExisting = true }))
667  //                using (var scope2 = new TempLinksTestScope(useSequences: true))
668  //                {
669  //                    scope1.Links.UseUnicode();
670  //                    scope2.Links.UseUnicode();
671  //
672  //                    //var compressor1 = new Compressor(scope1.Links.Unsync, scope1.Sequences);
673  //                    var compressor1 = scope1.Sequences;
674  //                    var compressor2 = scope2.Sequences;
675  //
676  //                    var compressed1 = new ulong[arrays.Length];
677  //                    var compressed2 = new ulong[arrays.Length];
678  //
679  //                    var sw1 = Stopwatch.StartNew();
```

```
680  //
681  //                    var START = 0;
682  //                    var END = arrays.Length;
683  //
684  //                    // Collisions proved (cannot be solved by max doublet comparison, no stable
     rule)
685  //                    // Stability issue starts at 10001 or 11000
686  //                    //for (int i = START; i < END; i++)
687  //                    //{
688  //                    //    var first = compressor1.Compress(arrays[i]);
689  //                    //    var second = compressor1.Compress(arrays[i]);
690  //
691  //                    //    if (first == second)
692  //                    //        compressed1[i] = first;
693  //                    //    else
694  //                    //    {
695  //                    //        // TODO: Find a solution for this case
696  //                    //    }
697  //                    //}
698  //
699  //                    for (int i = START; i < END; i++)
700  //                    {
701  //                        var first = compressor1.Create(arrays[i].ShiftRight());
702  //                        var second = compressor1.Create(arrays[i].ShiftRight());
703  //
704  //                        if (first == second)
705  //                        {
706  //                            compressed1[i] = first;
707  //                        }
708  //                        else
709  //                        {
710  //                            // TODO: Find a solution for this case
711  //                        }
712  //                    }
713  //
714  //                    var elapsed1 = sw1.Elapsed;
715  //
716  //                    var balancedVariantConverter = new
     BalancedVariantConverter<ulong>(scope2.Links);
717  //
718  //                    var sw2 = Stopwatch.StartNew();
719  //
720  //                    for (int i = START; i < END; i++)
721  //                    {
722  //                        var first = balancedVariantConverter.Convert(arrays[i]);
723  //                        var second = balancedVariantConverter.Convert(arrays[i]);
724  //
725  //                        if (first == second)
726  //                        {
727  //                            compressed2[i] = first;
728  //                        }
729  //                    }
730  //
731  //                    var elapsed2 = sw2.Elapsed;
732  //
733  //                    Debug.WriteLine($"Compressor: {elapsed1}, Balanced sequence creator:
     {elapsed2}");
734  //
735  //                    Assert.True(elapsed1 > elapsed2);
736  //
737  //                    // Checks
738  //                    for (int i = START; i < END; i++)
739  //                    {
740  //                        var sequence1 = compressed1[i];
741  //                        var sequence2 = compressed2[i];
742  //
743  //                        if (sequence1 != _constants.Null && sequence2 != _constants.Null)
744  //                        {
745  //                            var decompress1 = UnicodeMap.FromSequenceLinkToString(sequence1,
     scope1.Links);
746  //
747  //                            var decompress2 = UnicodeMap.FromSequenceLinkToString(sequence2,
     scope2.Links);
748  //
749  //                            //var structure1 = scope1.Links.FormatStructure(sequence1, link =>
     link.IsPartialPoint());
```

```
750  //                            //var structure2 = scope2.Links.FormatStructure(sequence2, link =>
     //    link.IsPartialPoint());
751  //
752  //                            //if (sequence1 != Constants.Null && sequence2 != Constants.Null &&
     //    arrays[i].Length > 3)
753  //                            //    Assert.False(structure1 == structure2);
754  //
755  //                            Assert.True(strings[i] == decompress1 && decompress1 == decompress2);
756  //                        }
757  //                    }
758  //
759  //                    Assert.True((int)(scope1.Links.Count() - UnicodeMap.MapSize) <
     //    totalCharacters);
760  //                    Assert.True((int)(scope2.Links.Count() - UnicodeMap.MapSize) <
     //    totalCharacters);
761  //
762  //                    Debug.WriteLine($"{(double)(scope1.Links.Count() - UnicodeMap.MapSize) /
     //    totalCharacters} | {(double)(scope2.Links.Count() - UnicodeMap.MapSize) / totalCharacters}");
763  //
764  //                    Assert.True(scope1.Links.Count() <= scope2.Links.Count());
765  //
766  //                    //compressor1.ValidateFrequencies();
767  //                }
768  //            }
769  //
770  //            [Fact]
771  //            public static void RundomNumbersCompressionQualityTest()
772  //            {
773  //                const ulong N = 500;
774  //
775  //                //const ulong minNumbers = 10000;
776  //                //const ulong maxNumbers = 20000;
777  //
778  //                //var strings = new List<string>();
779  //
780  //                //for (ulong i = 0; i < N; i++)
781  //                //    strings.Add(RandomHelpers.DefaultFactory.NextUInt64(minNumbers,
     //    maxNumbers).ToString());
782  //
783  //                var strings = new List<string>();
784  //
785  //                for (ulong i = 0; i < N; i++)
786  //                {
787  //                    strings.Add(RandomHelpers.Default.NextUInt64().ToString());
788  //                }
789  //
790  //                strings = strings.Distinct().ToList();
791  //
792  //                var arrays = strings.Select(UnicodeMap.FromStringToLinkArray).ToArray();
793  //                var totalCharacters = arrays.Select(x => x.Length).Sum();
794  //
795  //                using (var scope1 = new TempLinksTestScope(useSequences: true, sequencesOptions:
     //    new SequencesOptions<ulong> { UseCompression = true,
     //    EnforceSingleSequenceVersionOnWriteBasedOnExisting = true }))
796  //                using (var scope2 = new TempLinksTestScope(useSequences: true))
797  //                {
798  //                    scope1.Links.UseUnicode();
799  //                    scope2.Links.UseUnicode();
800  //
801  //                    var compressor1 = scope1.Sequences;
802  //                    var compressor2 = scope2.Sequences;
803  //
804  //                    var compressed1 = new ulong[arrays.Length];
805  //                    var compressed2 = new ulong[arrays.Length];
806  //
807  //                    var sw1 = Stopwatch.StartNew();
808  //
809  //                    var START = 0;
810  //                    var END = arrays.Length;
811  //
812  //                    for (int i = START; i < END; i++)
813  //                    {
814  //                        compressed1[i] = compressor1.Create(arrays[i].ShiftRight());
815  //                    }
816  //
817  //                    var elapsed1 = sw1.Elapsed;
818  //
```

```
819  //                        var balancedVariantConverter = new
     ↪  BalancedVariantConverter<ulong>(scope2.Links);
820  //
821  //                        var sw2 = Stopwatch.StartNew();
822  //
823  //                        for (int i = START; i < END; i++)
824  //                        {
825  //                            compressed2[i] = balancedVariantConverter.Convert(arrays[i]);
826  //                        }
827  //
828  //                        var elapsed2 = sw2.Elapsed;
829  //
830  //                        Debug.WriteLine($"Compressor: {elapsed1}, Balanced sequence creator:
     ↪  {elapsed2}");
831  //
832  //                        Assert.True(elapsed1 > elapsed2);
833  //
834  //                        // Checks
835  //                        for (int i = START; i < END; i++)
836  //                        {
837  //                            var sequence1 = compressed1[i];
838  //                            var sequence2 = compressed2[i];
839  //
840  //                            if (sequence1 != _constants.Null && sequence2 != _constants.Null)
841  //                            {
842  //                                var decompress1 = UnicodeMap.FromSequenceLinkToString(sequence1,
     ↪  scope1.Links);
843  //
844  //                                var decompress2 = UnicodeMap.FromSequenceLinkToString(sequence2,
     ↪  scope2.Links);
845  //
846  //                                Assert.True(strings[i] == decompress1 && decompress1 == decompress2);
847  //                            }
848  //                        }
849  //
850  //                        Assert.True((int)(scope1.Links.Count() - UnicodeMap.MapSize) <
     ↪  totalCharacters);
851  //                        Assert.True((int)(scope2.Links.Count() - UnicodeMap.MapSize) <
     ↪  totalCharacters);
852  //
853  //                        Debug.WriteLine($"{(double)(scope1.Links.Count() - UnicodeMap.MapSize) /
     ↪  totalCharacters} | {(double)(scope2.Links.Count() - UnicodeMap.MapSize) / totalCharacters}");
854  //
855  //                        // Can be worse than balanced variant
856  //                        //Assert.True(scope1.Links.Count() <= scope2.Links.Count());
857  //
858  //                        //compressor1.ValidateFrequencies();
859  //                    }
860  //                }
861  //
862  //            [Fact]
863  //            public static void AllTreeBreakDownAtSequencesCreationBugTest()
864  //            {
865  //                // Made out of AllPossibleConnectionsTest test.
866  //
867  //                //const long sequenceLength = 5; //100% bug
868  //                const long sequenceLength = 4; //100% bug
869  //                //const long sequenceLength = 3; //100% _no_bug (ok)
870  //
871  //                using (var scope = new TempLinksTestScope(useSequences: true))
872  //                {
873  //                    var links = scope.Links;
874  //                    var sequences = scope.Sequences;
875  //
876  //                    var sequence = new ulong[sequenceLength];
877  //                    for (var i = 0; i < sequenceLength; i++)
878  //                    {
879  //                        sequence[i] = links.Create();
880  //                    }
881  //
882  //                    var createResults = sequences.CreateAllVariants2(sequence);
883  //
884  //                    Global.Trash = createResults;
885  //
886  //                    for (var i = 0; i < sequenceLength; i++)
887  //                    {
888  //                        links.Delete(sequence[i]);
```

```csharp
            //                    }
            //                }
            //            }
            //
            //            [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]);
            //                    }
            //                }
            //            }
            //
            //            [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++)
            //                    {
```

```
965  //                          sequence[i] = links.Create();
966  //                      }
967  //
968  //                      var createResults = sequences.CreateAllVariants2(sequence);
969  //
970  //                      //var reverseResults =
     ↪  sequences.CreateAllVariants2(sequence.Reverse().ToArray());
971  //
972  //                      for (var i = 0; i < 1; i++)
973  //                      {
974  //                          var linksTotalUsages1 = new ulong[links.Count() + 1];
975  //
976  //                          sequences.CalculateAllUsages(linksTotalUsages1);
977  //
978  //                          var linksTotalUsages2 = new ulong[links.Count() + 1];
979  //
980  //                          sequences.CalculateAllUsages2(linksTotalUsages2);
981  //
982  //                          var intersection1 =
     ↪  linksTotalUsages1.Intersect(linksTotalUsages2).ToList();
983  //                          Assert.True(intersection1.Count == linksTotalUsages2.Length);
984  //                      }
985  //
986  //                      for (var i = 0; i < sequenceLength; i++)
987  //                      {
988  //                          links.Delete(sequence[i]);
989  //                      }
990  //                  }
991  //              }
992  //          }
993  // }
```

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

```
1   // using System.IO;
2   // using Platform.Disposables;
3   // using Platform.Data.Doublets.Sequences;
4   // using Platform.Data.Doublets.Decorators;
5   // using Platform.Data.Doublets.Memory.United.Specific;
6   // using Platform.Data.Doublets.Memory.Split.Specific;
7   // using Platform.Memory;
8   //
9   // namespace Platform.Data.Doublets.Sequences.Tests
10  // {
11  //     public class TempLinksTestScope : DisposableBase
12  //     {
13  //         public ILinks<ulong> MemoryAdapter { get; }
14  //         public SynchronizedLinks<ulong> Links { get; }
15  //         public Sequences Sequences { get; }
16  //         public string TempFilename { get; }
17  //         public string TempTransactionLogFilename { get; }
18  //         private readonly bool _deleteFiles;
19  //
20  //         public TempLinksTestScope(bool deleteFiles = true, bool useSequences = false, bool
     ↪  useLog = false) : this(new SequencesOptions<ulong>(), deleteFiles, useSequences, useLog) { }
21  //
22  //         public TempLinksTestScope(SequencesOptions<ulong> sequencesOptions, bool deleteFiles
     ↪  = true, bool useSequences = false, bool useLog = false)
23  //         {
24  //             _deleteFiles = deleteFiles;
25  //             TempFilename = Path.GetTempFileName();
26  //             TempTransactionLogFilename = Path.GetTempFileName();
27  //             //var coreMemoryAdapter = new UInt64UnitedMemoryLinks(TempFilename);
28  //             var coreMemoryAdapter = new UInt64SplitMemoryLinks(new
     ↪  FileMappedResizableDirectMemory(TempFilename), new
     ↪  FileMappedResizableDirectMemory(Path.ChangeExtension(TempFilename, "indexes")),
     ↪  UInt64SplitMemoryLinks.DefaultLinksSizeStep, new LinksConstants<ulong>(),
     ↪  Memory.IndexTreeType.Default, useLinkedList: true);
29  //             MemoryAdapter = useLog ? (ILinks<ulong>)new
     ↪  UInt64LinksTransactionsLayer(coreMemoryAdapter, TempTransactionLogFilename) :
     ↪  coreMemoryAdapter;
30  //             Links = new SynchronizedLinks<ulong>(new UInt64Links(MemoryAdapter));
31  //             if (useSequences)
32  //             {
33  //                 Sequences = new Sequences(Links, sequencesOptions);
34  //             }
35  //         }
36  //
```

```
37  //              protected override void Dispose(bool manual, bool wasDisposed)
38  //              {
39  //                  if (!wasDisposed)
40  //                  {
41  //                      Links.Unsync.DisposeIfPossible();
42  //                      if (_deleteFiles)
43  //                      {
44  //                          DeleteFiles();
45  //                      }
46  //                  }
47  //              }
48  //
49  //              public void DeleteFiles()
50  //              {
51  //                  File.Delete(TempFilename);
52  //                  File.Delete(TempTransactionLogFilename);
53  //              }
54  //          }
55  // }
```

1.70 ./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;
6   // using Platform.Setters;
7   // using Platform.Converters;
8   //
9   // namespace Platform.Data.Doublets.Sequences.Tests
10  // {
11  //     public static class TestExtensions
12  //     {
13  //         public static void TestCRUDOperations<T>(this ILinks<T> links)
14  //         {
15  //             var constants = links.Constants;
16  //
17  //             var equalityComparer = EqualityComparer<T>.Default;
18  //
19  //             var zero = default(T);
20  //             var one = Arithmetic.Increment(zero);
21  //
22  //             // Create Link
23  //             Assert.True(equalityComparer.Equals(links.Count(), zero));
24  //
25  //             var setter = new Setter<T>(constants.Null);
26  //             links.Each(constants.Any, constants.Any, setter.SetAndReturnTrue);
27  //
28  //             Assert.True(equalityComparer.Equals(setter.Result, constants.Null));
29  //
30  //             var linkAddress = links.Create();
31  //
32  //             var link = new Link<T>(links.GetLink(linkAddress));
33  //
34  //             Assert.True(link.Count == 3);
35  //             Assert.True(equalityComparer.Equals(link.Index, linkAddress));
36  //             Assert.True(equalityComparer.Equals(link.Source, constants.Null));
37  //             Assert.True(equalityComparer.Equals(link.Target, constants.Null));
38  //
39  //             Assert.True(equalityComparer.Equals(links.Count(), one));
40  //
41  //             // Get first link
42  //             setter = new Setter<T>(constants.Null);
43  //             links.Each(constants.Any, constants.Any, setter.SetAndReturnFalse);
44  //
45  //             Assert.True(equalityComparer.Equals(setter.Result, linkAddress));
46  //
47  //             // Update link to reference itself
48  //             links.Update(linkAddress, linkAddress, linkAddress);
49  //
50  //             link = new Link<T>(links.GetLink(linkAddress));
51  //
52  //             Assert.True(equalityComparer.Equals(link.Source, linkAddress));
53  //             Assert.True(equalityComparer.Equals(link.Target, linkAddress));
54  //
55  //             // Update link to reference null (prepare for delete)
56  //             var updated = links.Update(linkAddress, constants.Null, constants.Null);
57  //
```

```
58  //                    Assert.True(equalityComparer.Equals(updated, linkAddress));
59  //
60  //                    link = new Link<T>(links.GetLink(linkAddress));
61  //
62  //                    Assert.True(equalityComparer.Equals(link.Source, constants.Null));
63  //                    Assert.True(equalityComparer.Equals(link.Target, constants.Null));
64  //
65  //                    // Delete link
66  //                    links.Delete(linkAddress);
67  //
68  //                    Assert.True(equalityComparer.Equals(links.Count(), zero));
69  //
70  //                    setter = new Setter<T>(constants.Null);
71  //                    links.Each(constants.Any, constants.Any, setter.SetAndReturnTrue);
72  //
73  //                    Assert.True(equalityComparer.Equals(setter.Result, constants.Null));
74  //                }
75  //
76  //          public static void TestRawNumbersCRUDOperations<T>(this ILinks<T> links)
77  //                {
78  //                    // Constants
79  //                    var constants = links.Constants;
80  //                    var equalityComparer = EqualityComparer<T>.Default;
81  //
82  //                    var zero = default(T);
83  //                    var one = Arithmetic.Increment(zero);
84  //                    var two = Arithmetic.Increment(one);
85  //
86  //                    var h106E = new Hybrid<T>(106L, isExternal: true);
87  //                    var h107E = new Hybrid<T>(-char.ConvertFromUtf32(107)[0]);
88  //                    var h108E = new Hybrid<T>(-108L);
89  //
90  //                    Assert.Equal(106L, h106E.AbsoluteValue);
91  //                    Assert.Equal(107L, h107E.AbsoluteValue);
92  //                    Assert.Equal(108L, h108E.AbsoluteValue);
93  //
94  //                    // Create Link (External -> External)
95  //                    var linkAddress1 = links.Create();
96  //
97  //                    links.Update(linkAddress1, h106E, h108E);
98  //
99  //                    var link1 = new Link<T>(links.GetLink(linkAddress1));
100 //
101 //                    Assert.True(equalityComparer.Equals(link1.Source, h106E));
102 //                    Assert.True(equalityComparer.Equals(link1.Target, h108E));
103 //
104 //                    // Create Link (Internal -> External)
105 //                    var linkAddress2 = links.Create();
106 //
107 //                    links.Update(linkAddress2, linkAddress1, h108E);
108 //
109 //                    var link2 = new Link<T>(links.GetLink(linkAddress2));
110 //
111 //                    Assert.True(equalityComparer.Equals(link2.Source, linkAddress1));
112 //                    Assert.True(equalityComparer.Equals(link2.Target, h108E));
113 //
114 //                    // Create Link (Internal -> Internal)
115 //                    var linkAddress3 = links.Create();
116 //
117 //                    links.Update(linkAddress3, linkAddress1, linkAddress2);
118 //
119 //                    var link3 = new Link<T>(links.GetLink(linkAddress3));
120 //
121 //                    Assert.True(equalityComparer.Equals(link3.Source, linkAddress1));
122 //                    Assert.True(equalityComparer.Equals(link3.Target, linkAddress2));
123 //
124 //                    // Search for created link
125 //                    var setter1 = new Setter<T>(constants.Null);
126 //                    links.Each(h106E, h108E, setter1.SetAndReturnFalse);
127 //
128 //                    Assert.True(equalityComparer.Equals(setter1.Result, linkAddress1));
129 //
130 //                    // Search for nonexistent link
131 //                    var setter2 = new Setter<T>(constants.Null);
132 //                    links.Each(h106E, h107E, setter2.SetAndReturnFalse);
133 //
134 //                    Assert.True(equalityComparer.Equals(setter2.Result, constants.Null));
135 //
```

```
136  //                    // Update link to reference null (prepare for delete)
137  //                    var updated = links.Update(linkAddress3, constants.Null, constants.Null);
138  //
139  //                    Assert.True(equalityComparer.Equals(updated, linkAddress3));
140  //
141  //                    link3 = new Link<T>(links.GetLink(linkAddress3));
142  //
143  //                    Assert.True(equalityComparer.Equals(link3.Source, constants.Null));
144  //                    Assert.True(equalityComparer.Equals(link3.Target, constants.Null));
145  //
146  //                    // Delete link
147  //                    links.Delete(linkAddress3);
148  //
149  //                    Assert.True(equalityComparer.Equals(links.Count(), two));
150  //
151  //                    var setter3 = new Setter<T>(constants.Null);
152  //                    links.Each(constants.Any, constants.Any, setter3.SetAndReturnTrue);
153  //
154  //                    Assert.True(equalityComparer.Equals(setter3.Result, linkAddress2));
155  //        }
156  //
157  //        public static void TestMultipleRandomCreationsAndDeletions<TLinkAddress>(this
    ↪  ILinks<TLinkAddress> links, int maximumOperationsPerCycle)
158  //        {
159  //            var comparer = Comparer<TLinkAddress>.Default;
160  //            var addressToUInt64Converter = CheckedConverter<TLinkAddress, ulong>.Default;
161  //            var uInt64ToAddressConverter = CheckedConverter<ulong, TLinkAddress>.Default;
162  //            for (var N = 1; N < maximumOperationsPerCycle; N++)
163  //            {
164  //                var random = new System.Random(N);
165  //                var created = 0UL;
166  //                var deleted = 0UL;
167  //                for (var i = 0; i < N; i++)
168  //                {
169  //                    var linksCount = addressToUInt64Converter.Convert(links.Count());
170  //                    var createPoint = random.NextBoolean();
171  //                    if (linksCount >= 2 && createPoint)
172  //                    {
173  //                        var linksAddressRange = new Range<ulong>(1, linksCount);
174  //                        TLinkAddress source =
    ↪  uInt64ToAddressConverter.Convert(random.NextUInt64(linksAddressRange));
175  //                        TLinkAddress target =
    ↪  uInt64ToAddressConverter.Convert(random.NextUInt64(linksAddressRange)); //-V3086
176  //                        var resultLink = links.GetOrCreate(source, target);
177  //                        if (comparer.Compare(resultLink,
    ↪  uInt64ToAddressConverter.Convert(linksCount)) > 0)
178  //                        {
179  //                            created++;
180  //                        }
181  //                    }
182  //                    else
183  //                    {
184  //                        links.Create();
185  //                        created++;
186  //                    }
187  //                }
188  //                Assert.True(created == addressToUInt64Converter.Convert(links.Count()));
189  //                for (var i = 0; i < N; i++)
190  //                {
191  //                    TLinkAddress link = uInt64ToAddressConverter.Convert((ulong)i + 1UL);
192  //                    if (links.Exists(link))
193  //                    {
194  //                        links.Delete(link);
195  //                        deleted++;
196  //                    }
197  //                }
198  //                Assert.True(addressToUInt64Converter.Convert(links.Count()) == 0L);
199  //            }
200  //        }
201  //    }
202  // }
```

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

```
1  // using System;
2  // using System.Collections.Generic;
3  // using System.Diagnostics;
4  // using System.IO;
```

```csharp
// 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
// {
//     public static class UInt64LinksTests
//     {
//         private static readonly LinksConstants<ulong> _constants =
//   Default<LinksConstants<ulong>>.Instance;
//         private const long Iterations = 10 * 1024;
//
//         #region Concept
//
//         [Fact]
//         public static void MultipleCreateAndDeleteTest()
//         {
//             using (var scope = new Scope<Types<HeapResizableDirectMemory,
//   UInt64UnitedMemoryLinks>>())
//             {
//                 new
//   UInt64Links(scope.Use<ILinks<ulong>>()).TestMultipleRandomCreationsAndDeletions(100);
//             }
//         }
//
//         [Fact]
//         public static void CascadeUpdateTest()
//         {
//             var itself = _constants.Itself;
//             using (var scope = new TempLinksTestScope(useLog: true))
//             {
//                 var links = scope.Links;
//
//                 var l1 = links.Create();
//                 var l2 = links.Create();
//
//                 l2 = links.Update(l2, l2, l1, l2);
//
//                 links.CreateAndUpdate(l2, itself);
//                 links.CreateAndUpdate(l2, itself);
//
//                 l2 = links.Update(l2, l1);
//
//                 links.Delete(l2);
//
//                 Global.Trash = links.Count();
//
//                 links.Unsync.DisposeIfPossible(); // Close links to access log
//
//                 Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(s⌋
//   cope.TempTransactionLogFilename);
//             }
//         }
//
//         [Fact]
//         public static void BasicTransactionLogTest()
//         {
//             using (var scope = new TempLinksTestScope(useLog: true))
//             {
//                 var links = scope.Links;
//                 var l1 = links.Create();
//                 var l2 = links.Create();
//
```

```
 78  //                        Global.Trash = links.Update(l2, l2, l1, l2);
 79  //
 80  //                        links.Delete(l1);
 81  //
 82  //                        links.Unsync.DisposeIfPossible(); // Close links to access log
 83  //
 84  //                        Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(s
     ↪  cope.TempTransactionLogFilename);
 85  //                    }
 86  //                }
 87  //
 88  //                [Fact]
 89  //                public static void TransactionAutoRevertedTest()
 90  //                {
 91  //                    // Auto Reverted (Because no commit at transaction)
 92  //                    using (var scope = new TempLinksTestScope(useLog: true))
 93  //                    {
 94  //                        var links = scope.Links;
 95  //                        var transactionsLayer = (UInt64LinksTransactionsLayer)scope.MemoryAdapter;
 96  //                        using (var transaction = transactionsLayer.BeginTransaction())
 97  //                        {
 98  //                            var l1 = links.Create();
 99  //                            var l2 = links.Create();
100  //
101  //                            links.Update(l2, l2, l1, l2);
102  //                        }
103  //
104  //                        Assert.Equal(0UL, links.Count());
105  //
106  //                        links.Unsync.DisposeIfPossible();
107  //
108  //                        var transitions = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition
     ↪  >(scope.TempTransactionLogFilename);
109  //                        Assert.Single(transitions);
110  //                    }
111  //                }
112  //
113  //                [Fact]
114  //                public static void TransactionUserCodeErrorNoDataSavedTest()
115  //                {
116  //                    // User Code Error (Autoreverted), no data saved
117  //                    var itself = _constants.Itself;
118  //
119  //                    TempLinksTestScope lastScope = null;
120  //                    try
121  //                    {
122  //                        using (var scope = lastScope = new TempLinksTestScope(deleteFiles: false,
     ↪  useLog: true))
123  //                        {
124  //                            var links = scope.Links;
125  //                            var transactionsLayer =
     ↪  (UInt64LinksTransactionsLayer)((LinksDisposableDecoratorBase<ulong>)links.Unsync).Links;
126  //                            using (var transaction = transactionsLayer.BeginTransaction())
127  //                            {
128  //                                var l1 = links.CreateAndUpdate(itself, itself);
129  //                                var l2 = links.CreateAndUpdate(itself, itself);
130  //
131  //                                l2 = links.Update(l2, l2, l1, l2);
132  //
133  //                                links.CreateAndUpdate(l2, itself);
134  //                                links.CreateAndUpdate(l2, itself);
135  //
136  //                                //Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Tra
     ↪  nsition>(scope.TempTransactionLogFilename);
137  //
138  //                                l2 = links.Update(l2, l1);
139  //
140  //                                links.Delete(l2);
141  //
142  //                                ExceptionThrower();
143  //
144  //                                transaction.Commit();
145  //                            }
146  //
147  //                            Global.Trash = links.Count();
148  //                        }
149  //                    }
```

```
150  //              catch
151  //              {
152  //                  Assert.False(lastScope == null);
153  //
154  //                  var transitions = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition
     ↪  >(lastScope.TempTransactionLogFilename);
155  //
156  //                  Assert.True(transitions.Length == 1 && transitions[0].Before.IsNull() &&
     ↪  transitions[0].After.IsNull());
157  //
158  //                  lastScope.DeleteFiles();
159  //              }
160  //          }
161  //
162  //          [Fact]
163  //          public static void TransactionUserCodeErrorSomeDataSavedTest()
164  //          {
165  //              // User Code Error (Autoreverted), some data saved
166  //              var itself = _constants.Itself;
167  //
168  //              TempLinksTestScope lastScope = null;
169  //              try
170  //              {
171  //                  ulong l1;
172  //                  ulong l2;
173  //
174  //                  using (var scope = new TempLinksTestScope(useLog: true))
175  //                  {
176  //                      var links = scope.Links;
177  //                      l1 = links.CreateAndUpdate(itself, itself);
178  //                      l2 = links.CreateAndUpdate(itself, itself);
179  //
180  //                      l2 = links.Update(l2, l2, l1, l2);
181  //
182  //                      links.CreateAndUpdate(l2, itself);
183  //                      links.CreateAndUpdate(l2, itself);
184  //
185  //                      links.Unsync.DisposeIfPossible();
186  //
187  //                      Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transitio
     ↪  n>(scope.TempTransactionLogFilename);
188  //                  }
189  //
190  //                  using (var scope = lastScope = new TempLinksTestScope(deleteFiles: false,
     ↪  useLog: true))
191  //                  {
192  //                      var links = scope.Links;
193  //                      var transactionsLayer = (UInt64LinksTransactionsLayer)links.Unsync;
194  //                      using (var transaction = transactionsLayer.BeginTransaction())
195  //                      {
196  //                          l2 = links.Update(l2, l1);
197  //
198  //                          links.Delete(l2);
199  //
200  //                          ExceptionThrower();
201  //
202  //                          transaction.Commit();
203  //                      }
204  //
205  //                      Global.Trash = links.Count();
206  //                  }
207  //              }
208  //              catch
209  //              {
210  //                  Assert.False(lastScope == null);
211  //
212  //                  Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(l
     ↪  astScope.TempTransactionLogFilename);
213  //
214  //                  lastScope.DeleteFiles();
215  //              }
216  //          }
217  //
218  //          [Fact]
219  //          public static void TransactionCommit()
220  //          {
221  //              var itself = _constants.Itself;
```

```
222  //
223  //                  var tempDatabaseFilename = Path.GetTempFileName();
224  //                  var tempTransactionLogFilename = Path.GetTempFileName();
225  //
226  //                  // Commit
227  //                  using (var memoryAdapter = new UInt64LinksTransactionsLayer(new
     ↪  UInt64UnitedMemoryLinks(tempDatabaseFilename), tempTransactionLogFilename))
228  //                  using (var links = new UInt64Links(memoryAdapter))
229  //                  {
230  //                      using (var transaction = memoryAdapter.BeginTransaction())
231  //                      {
232  //                          var l1 = links.CreateAndUpdate(itself, itself);
233  //                          var l2 = links.CreateAndUpdate(itself, itself);
234  //
235  //                          Global.Trash = links.Update(l2, l2, l1, l2);
236  //
237  //                          links.Delete(l1);
238  //
239  //                          transaction.Commit();
240  //                      }
241  //
242  //                      Global.Trash = links.Count();
243  //                  }
244  //
245  //                  Global.Trash =
     ↪  FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(tempTransactionLogFilename);
246  //              }
247  //
248  //          [Fact]
249  //          public static void TransactionDamage()
250  //          {
251  //                  var itself = _constants.Itself;
252  //
253  //                  var tempDatabaseFilename = Path.GetTempFileName();
254  //                  var tempTransactionLogFilename = Path.GetTempFileName();
255  //
256  //                  // Commit
257  //                  using (var memoryAdapter = new UInt64LinksTransactionsLayer(new
     ↪  UInt64UnitedMemoryLinks(tempDatabaseFilename), tempTransactionLogFilename))
258  //                  using (var links = new UInt64Links(memoryAdapter))
259  //                  {
260  //                      using (var transaction = memoryAdapter.BeginTransaction())
261  //                      {
262  //                          var l1 = links.CreateAndUpdate(itself, itself);
263  //                          var l2 = links.CreateAndUpdate(itself, itself);
264  //
265  //                          Global.Trash = links.Update(l2, l2, l1, l2);
266  //
267  //                          links.Delete(l1);
268  //
269  //                          transaction.Commit();
270  //                      }
271  //
272  //                      Global.Trash = links.Count();
273  //                  }
274  //
275  //                  Global.Trash =
     ↪  FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(tempTransactionLogFilename);
276  //
277  //                  // Damage database
278  //
279  //                  FileHelpers.WriteFirst(tempTransactionLogFilename, new
     ↪  UInt64LinksTransactionsLayer.Transition(new UniqueTimestampFactory(), 555));
280  //
281  //                  // Try load damaged database
282  //                  try
283  //                  {
284  //                      // TODO: Fix
285  //                      using (var memoryAdapter = new UInt64LinksTransactionsLayer(new
     ↪  UInt64UnitedMemoryLinks(tempDatabaseFilename), tempTransactionLogFilename))
286  //                      using (var links = new UInt64Links(memoryAdapter))
287  //                      {
288  //                          Global.Trash = links.Count();
289  //                      }
290  //                  }
291  //                  catch (NotSupportedException ex)
292  //                  {
```

```
293  //                   Assert.True(ex.Message == "Database is damaged, autorecovery is not supported
     ↪ yet.");
294  //               }
295  //
296  //               Global.Trash =
     ↪ FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(tempTransactionLogFilename);
297  //
298  //               File.Delete(tempDatabaseFilename);
299  //               File.Delete(tempTransactionLogFilename);
300  //           }
301  //
302  //           [Fact]
303  //           public static void Bug1Test()
304  //           {
305  //               var tempDatabaseFilename = Path.GetTempFileName();
306  //               var tempTransactionLogFilename = Path.GetTempFileName();
307  //
308  //               var itself = _constants.Itself;
309  //
310  //               // User Code Error (Autoreverted), some data saved
311  //               try
312  //               {
313  //                   ulong l1;
314  //                   ulong l2;
315  //
316  //                   using (var memory = new UInt64UnitedMemoryLinks(tempDatabaseFilename))
317  //                   using (var memoryAdapter = new UInt64LinksTransactionsLayer(memory,
     ↪ tempTransactionLogFilename))
318  //                   using (var links = new UInt64Links(memoryAdapter))
319  //                   {
320  //                       l1 = links.CreateAndUpdate(itself, itself);
321  //                       l2 = links.CreateAndUpdate(itself, itself);
322  //
323  //                       l2 = links.Update(l2, l2, l1, l2);
324  //
325  //                       links.CreateAndUpdate(l2, itself);
326  //                       links.CreateAndUpdate(l2, itself);
327  //                   }
328  //
329  //                   Global.Trash =
     ↪ FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(tempTransactionLogFilename);
330  //
331  //                   using (var memory = new UInt64UnitedMemoryLinks(tempDatabaseFilename))
332  //                   using (var memoryAdapter = new UInt64LinksTransactionsLayer(memory,
     ↪ tempTransactionLogFilename))
333  //                   using (var links = new UInt64Links(memoryAdapter))
334  //                   {
335  //                       using (var transaction = memoryAdapter.BeginTransaction())
336  //                       {
337  //                           l2 = links.Update(l2, l1);
338  //
339  //                           links.Delete(l2);
340  //
341  //                           ExceptionThrower();
342  //
343  //                           transaction.Commit();
344  //                       }
345  //
346  //                       Global.Trash = links.Count();
347  //                   }
348  //               }
349  //               catch
350  //               {
351  //                   Global.Trash =
     ↪ FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(tempTransactionLogFilename);
352  //               }
353  //
354  //               File.Delete(tempDatabaseFilename);
355  //               File.Delete(tempTransactionLogFilename);
356  //           }
357  //           private static void ExceptionThrower() => throw new InvalidOperationException();
358  //
359  //           [Fact]
360  //           public static void PathsTest()
361  //           {
362  //               var source = _constants.SourcePart;
363  //               var target = _constants.TargetPart;
```

```
364  //
365  //                using (var scope = new TempLinksTestScope())
366  //                {
367  //                    var links = scope.Links;
368  //                    var l1 = links.CreatePoint();
369  //                    var l2 = links.CreatePoint();
370  //
371  //                    var r1 = links.GetByKeys(l1, source, target, source);
372  //                    var r2 = links.CheckPathExistance(l2, l2, l2, l2);
373  //                }
374  //            }
375  //
376  //            [Fact]
377  //            public static void RecursiveStringFormattingTest()
378  //            {
379  //                using (var scope = new TempLinksTestScope(useSequences: true))
380  //                {
381  //                    var links = scope.Links;
382  //                    var sequences = scope.Sequences; // TODO: Auto use sequences on Sequences
     ↪  getter.
383  //
384  //                    var a = links.CreatePoint();
385  //                    var b = links.CreatePoint();
386  //                    var c = links.CreatePoint();
387  //
388  //                    var ab = links.GetOrCreate(a, b);
389  //                    var cb = links.GetOrCreate(c, b);
390  //                    var ac = links.GetOrCreate(a, c);
391  //
392  //                    a = links.Update(a, c, b);
393  //                    b = links.Update(b, a, c);
394  //                    c = links.Update(c, a, b);
395  //
396  //                    Debug.WriteLine(links.FormatStructure(ab, link => link.IsFullPoint(), true));
397  //                    Debug.WriteLine(links.FormatStructure(cb, link => link.IsFullPoint(), true));
398  //                    Debug.WriteLine(links.FormatStructure(ac, link => link.IsFullPoint(), true));
399  //
400  //                    Assert.True(links.FormatStructure(cb, link => link.IsFullPoint(), true) ==
     ↪  "(5:(4:5 (6:5 4)) 6)");
401  //                    Assert.True(links.FormatStructure(ac, link => link.IsFullPoint(), true) ==
     ↪  "(6:(5:(4:5 6) 6) 4)");
402  //                    Assert.True(links.FormatStructure(ab, link => link.IsFullPoint(), true) ==
     ↪  "(4:(5:4 (6:5 4)) 6)");
403  //
404  //                    // 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)"
405  //
406  //                    Assert.True(sequences.SafeFormatSequence(cb, DefaultFormatter, false) ==
     ↪  "{{5}{5}{4}{6}}");
407  //                    Assert.True(sequences.SafeFormatSequence(ac, DefaultFormatter, false) ==
     ↪  "{{5}{6}{6}{4}}");
408  //                    Assert.True(sequences.SafeFormatSequence(ab, DefaultFormatter, false) ==
     ↪  "{{4}{5}{4}{6}}");
409  //                }
410  //            }
411  //            private static void DefaultFormatter(StringBuilder sb, ulong link)
412  //            {
413  //                sb.Append(link.ToString());
414  //            }
415  //
416  //            #endregion
417  //
418  //            #region Performance
419  //
420  //            /*
421  //            public static void RunAllPerformanceTests()
422  //            {
423  //                try
424  //                {
425  //                    links.TestLinksInSteps();
426  //                }
427  //                catch (Exception ex)
428  //                {
429  //                    ex.WriteToConsole();
430  //                }
431  //
432  //                return;
```

```
433  //
434  //                try
435  //                {
436  //                    //ThreadPool.SetMaxThreads(2, 2);
437  //
438  //                    // Запускаем все тесты дважды, чтобы первоначальная инициализация не повлияла
     ↪ на результат
439  //                    // Также это дополнительно помогает в отладке
440  //                    // Увеличивает вероятность попадания информации в кэши
441  //                    for (var i = 0; i < 10; i++)
442  //                    {
443  //                        //0 - 10 ГБ
444  //                        //Каждые 100 МБ срез цифр
445  //
446  //                        //links.TestGetSourceFunction();
447  //                        //links.TestGetSourceFunctionInParallel();
448  //                        //links.TestGetTargetFunction();
449  //                        //links.TestGetTargetFunctionInParallel();
450  //                        links.Create64BillionLinks();
451  //
452  //                        links.TestRandomSearchFixed();
453  //                        //links.Create64BillionLinksInParallel();
454  //                        links.TestEachFunction();
455  //                        //links.TestForeach();
456  //                        //links.TestParallelForeach();
457  //                    }
458  //
459  //                    links.TestDeletionOfAllLinks();
460  //
461  //                }
462  //                catch (Exception ex)
463  //                {
464  //                    ex.WriteToConsole();
465  //                }
466  //            }*/
467  //
468  //             /*
469  //            public static void TestLinksInSteps()
470  //            {
471  //                const long gibibyte = 1024 * 1024 * 1024;
472  //                const long mebibyte = 1024 * 1024;
473  //
474  //                var totalLinksToCreate = gibibyte /
     ↪ Platform.Links.Data.Core.Doublets.Links.LinkSizeInBytes;
475  //                var linksStep = 102 * mebibyte /
     ↪ Platform.Links.Data.Core.Doublets.Links.LinkSizeInBytes;
476  //
477  //                var creationMeasurements = new List<TimeSpan>();
478  //                var searchMeasuremets = new List<TimeSpan>();
479  //                var deletionMeasurements = new List<TimeSpan>();
480  //
481  //                GetBaseRandomLoopOverhead(linksStep);
482  //                GetBaseRandomLoopOverhead(linksStep);
483  //
484  //                var stepLoopOverhead = GetBaseRandomLoopOverhead(linksStep);
485  //
486  //                ConsoleHelpers.Debug("Step loop overhead: {0}.", stepLoopOverhead);
487  //
488  //                var loops = totalLinksToCreate / linksStep;
489  //
490  //                for (int i = 0; i < loops; i++)
491  //                {
492  //                    creationMeasurements.Add(Measure(() => links.RunRandomCreations(linksStep)));
493  //                    searchMeasuremets.Add(Measure(() => links.RunRandomSearches(linksStep)));
494  //
495  //                    Console.Write("\rC + S {0}/{1}", i + 1, loops);
496  //                }
497  //
498  //                ConsoleHelpers.Debug();
499  //
500  //                for (int i = 0; i < loops; i++)
501  //                {
502  //                    deletionMeasurements.Add(Measure(() => links.RunRandomDeletions(linksStep)));
503  //
504  //                    Console.Write("\rD {0}/{1}", i + 1, loops);
505  //                }
506  //
507  //                ConsoleHelpers.Debug();
```

```
//
//                ConsoleHelpers.Debug("C S D");
//
//                for (int i = 0; i < loops; i++)
//                {
//                    ConsoleHelpers.Debug("{0} {1} {2}", creationMeasurements[i],
//  searchMeasuremets[i], deletionMeasurements[i]);
//                }
//
//                ConsoleHelpers.Debug("C S D (no overhead)");
//
//                for (int i = 0; i < loops; i++)
//                {
//                    ConsoleHelpers.Debug("{0} {1} {2}", creationMeasurements[i] -
//  stepLoopOverhead, searchMeasuremets[i] - stepLoopOverhead, deletionMeasurements[i] -
//  stepLoopOverhead);
//                }
//
//                ConsoleHelpers.Debug("All tests done. Total links left in database: {0}.",
//  links.Total);
//            }
//
//        private static void CreatePoints(this Platform.Links.Data.Core.Doublets.Links links,
//  long amountToCreate)
//        {
//            for (long i = 0; i < amountToCreate; i++)
//                links.Create(0, 0);
//        }
//
//         private static TimeSpan GetBaseRandomLoopOverhead(long loops)
//         {
//             return Measure(() =>
//             {
//                 ulong maxValue = RandomHelpers.DefaultFactory.NextUInt64();
//                 ulong result = 0;
//                 for (long i = 0; i < loops; i++)
//                 {
//                     var source = RandomHelpers.DefaultFactory.NextUInt64(maxValue);
//                     var target = RandomHelpers.DefaultFactory.NextUInt64(maxValue);
//
//                     result += maxValue + source + target;
//                 }
//                 Global.Trash = result;
//             });
//         }
//          */
//
//        [Fact(Skip = "performance test")]
//        public static void GetSourceTest()
//        {
//            using (var scope = new TempLinksTestScope())
//            {
//                var links = scope.Links;
//                ConsoleHelpers.Debug("Testing GetSource function with {0} Iterations.",
//  Iterations);
//
//                ulong counter = 0;
//
//                //var firstLink = links.First();
//                // Создаём одну связь, из которой будет производить считывание
//                var firstLink = links.Create();
//
//                var sw = Stopwatch.StartNew();
//
//                // Тестируем саму функцию
//                for (ulong i = 0; i < Iterations; i++)
//                {
//                    counter += links.GetSource(firstLink);
//                }
//
//                var elapsedTime = sw.Elapsed;
//
//                var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;
//
//                // Удаляем связь, из которой производилось считывание
//                links.Delete(firstLink);
//
```

```
579  //                        ConsoleHelpers.Debug(
580  //                            "{0} Iterations of GetSource function done in {1} ({2} Iterations per
     ↪  second), counter result: {3}",
581  //                            Iterations, elapsedTime, (long)iterationsPerSecond, counter);
582  //                }
583  //            }
584  //
585  //            [Fact(Skip = "performance test")]
586  //            public static void GetSourceInParallel()
587  //            {
588  //                using (var scope = new TempLinksTestScope())
589  //                {
590  //                    var links = scope.Links;
591  //                    ConsoleHelpers.Debug("Testing GetSource function with {0} Iterations in
     ↪  parallel.", Iterations);
592  //
593  //                    long counter = 0;
594  //
595  //                    //var firstLink = links.First();
596  //                    var firstLink = links.Create();
597  //
598  //                    var sw = Stopwatch.StartNew();
599  //
600  //                    // Тестируем саму функцию
601  //                    Parallel.For(0, Iterations, x =>
602  //                    {
603  //                        Interlocked.Add(ref counter, (long)links.GetSource(firstLink));
604  //                        //Interlocked.Increment(ref counter);
605  //                    });
606  //
607  //                    var elapsedTime = sw.Elapsed;
608  //
609  //                    var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;
610  //
611  //                    links.Delete(firstLink);
612  //
613  //                    ConsoleHelpers.Debug(
614  //                        "{0} Iterations of GetSource function done in {1} ({2} Iterations per
     ↪  second), counter result: {3}",
615  //                        Iterations, elapsedTime, (long)iterationsPerSecond, counter);
616  //                }
617  //            }
618  //
619  //            [Fact(Skip = "performance test")]
620  //            public static void TestGetTarget()
621  //            {
622  //                using (var scope = new TempLinksTestScope())
623  //                {
624  //                    var links = scope.Links;
625  //                    ConsoleHelpers.Debug("Testing GetTarget function with {0} Iterations.",
     ↪  Iterations);
626  //
627  //                    ulong counter = 0;
628  //
629  //                    //var firstLink = links.First();
630  //                    var firstLink = links.Create();
631  //
632  //                    var sw = Stopwatch.StartNew();
633  //
634  //                    for (ulong i = 0; i < Iterations; i++)
635  //                    {
636  //                        counter += links.GetTarget(firstLink);
637  //                    }
638  //
639  //                    var elapsedTime = sw.Elapsed;
640  //
641  //                    var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;
642  //
643  //                    links.Delete(firstLink);
644  //
645  //                    ConsoleHelpers.Debug(
646  //                        "{0} Iterations of GetTarget function done in {1} ({2} Iterations per
     ↪  second), counter result: {3}",
647  //                        Iterations, elapsedTime, (long)iterationsPerSecond, counter);
648  //                }
649  //            }
650  //
```

```
651  //          [Fact(Skip = "performance test")]
652  //          public static void TestGetTargetInParallel()
653  //          {
654  //              using (var scope = new TempLinksTestScope())
655  //              {
656  //                  var links = scope.Links;
657  //                  ConsoleHelpers.Debug("Testing GetTarget function with {0} Iterations in
     ↪ parallel.", Iterations);
658  //
659  //                  long counter = 0;
660  //
661  //                  //var firstLink = links.First();
662  //                  var firstLink = links.Create();
663  //
664  //                  var sw = Stopwatch.StartNew();
665  //
666  //                  Parallel.For(0, Iterations, x =>
667  //                  {
668  //                      Interlocked.Add(ref counter, (long)links.GetTarget(firstLink));
669  //                      //Interlocked.Increment(ref counter);
670  //                  });
671  //
672  //                  var elapsedTime = sw.Elapsed;
673  //
674  //                  var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;
675  //
676  //                  links.Delete(firstLink);
677  //
678  //                  ConsoleHelpers.Debug(
679  //                      "{0} Iterations of GetTarget function done in {1} ({2} Iterations per
     ↪ second), counter result: {3}",
680  //                      Iterations, elapsedTime, (long)iterationsPerSecond, counter);
681  //              }
682  //          }
683  //
684  //          // TODO: Заполнить базу данных перед тестом
685  //          /*
686  //          [Fact]
687  //          public void TestRandomSearchFixed()
688  //          {
689  //              var tempFilename = Path.GetTempFileName();
690  //
691  //              using (var links = new Platform.Links.Data.Core.Doublets.Links(tempFilename,
     ↪ DefaultLinksSizeStep))
692  //              {
693  //                  long iterations = 64 * 1024 * 1024 /
     ↪ Platform.Links.Data.Core.Doublets.Links.LinkSizeInBytes;
694  //
695  //                  ulong counter = 0;
696  //                  var maxLink = links.Total;
697  //
698  //                  ConsoleHelpers.Debug("Testing Random Search with {0} Iterations.",
     ↪ iterations);
699  //
700  //                  var sw = Stopwatch.StartNew();
701  //
702  //                  for (var i = iterations; i > 0; i--)
703  //                  {
704  //                      var source =
     ↪ RandomHelpers.DefaultFactory.NextUInt64(LinksConstants.MinPossibleIndex, maxLink);
705  //                      var target =
     ↪ RandomHelpers.DefaultFactory.NextUInt64(LinksConstants.MinPossibleIndex, maxLink);
706  //
707  //                      counter += links.Search(source, target);
708  //                  }
709  //
710  //                  var elapsedTime = sw.Elapsed;
711  //
712  //                  var iterationsPerSecond = iterations / elapsedTime.TotalSeconds;
713  //
714  //                  ConsoleHelpers.Debug("{0} Iterations of Random Search done in {1} ({2}
     ↪ Iterations per second), c: {3}", iterations, elapsedTime, (long)iterationsPerSecond,
     ↪ counter);
715  //              }
716  //
717  //              File.Delete(tempFilename);
718  //          }*/
```

```csharp
//
//         [Fact(Skip = "useless: O(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);
//             }
//         }
//
//         [Fact(Skip = "useless: O(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.");
//
//                 var sw = Stopwatch.StartNew();
//
//                 links.Each(counter.IncrementAndReturnTrue);
//
//                 var elapsedTime = sw.Elapsed;
//
//                 var linksPerSecond = counter.Count / elapsedTime.TotalSeconds;
//
//                 ConsoleHelpers.Debug("{0} Iterations of Each's handler function done in {1}
//   ({2} links per second)",
//                     counter, elapsedTime, (long)linksPerSecond);
//             }
//         }
//
//         /*
//         [Fact]
//         public static void TestForeach()
//         {
//             var tempFilename = Path.GetTempFileName();
//
//             using (var links = new Platform.Links.Data.Core.Doublets.Links(tempFilename,
//   DefaultLinksSizeStep))
//             {
//                 ulong counter = 0;
//
//                 ConsoleHelpers.Debug("Testing foreach through links.");
//
//
```

```
791  //                      var sw = Stopwatch.StartNew();
792  //
793  //                      //foreach (var link in links)
794  //                      //{
795  //                      //    counter++;
796  //                      //}
797  //
798  //                      var elapsedTime = sw.Elapsed;
799  //
800  //                      var linksPerSecond = (double)counter / elapsedTime.TotalSeconds;
801  //
802  //                      ConsoleHelpers.Debug("{0} Iterations of Foreach's handler block done in {1}
     ↪  ({2} links per second)", counter, elapsedTime, (long)linksPerSecond);
803  //                  }
804  //
805  //                  File.Delete(tempFilename);
806  //              }
807  //          */
808  //
809  //          /*
810  //          [Fact]
811  //          public static void TestParallelForeach()
812  //          {
813  //              var tempFilename = Path.GetTempFileName();
814  //
815  //              using (var links = new Platform.Links.Data.Core.Doublets.Links(tempFilename,
     ↪  DefaultLinksSizeStep))
816  //              {
817  //
818  //                      long counter = 0;
819  //
820  //                      ConsoleHelpers.Debug("Testing parallel foreach through links.");
821  //
822  //                      var sw = Stopwatch.StartNew();
823  //
824  //                      //Parallel.ForEach((IEnumerable<ulong>)links, x =>
825  //                      //{
826  //                      //    Interlocked.Increment(ref counter);
827  //                      //});
828  //
829  //                      var elapsedTime = sw.Elapsed;
830  //
831  //                      var linksPerSecond = (double)counter / elapsedTime.TotalSeconds;
832  //
833  //                      ConsoleHelpers.Debug("{0} Iterations of Parallel Foreach's handler block done
     ↪  in {1} ({2} links per second)", counter, elapsedTime, (long)linksPerSecond);
834  //                  }
835  //
836  //                  File.Delete(tempFilename);
837  //              }
838  //          */
839  //
840  //          [Fact(Skip = "performance test")]
841  //          public static void Create64BillionLinks()
842  //          {
843  //              using (var scope = new TempLinksTestScope())
844  //              {
845  //                  var links = scope.Links;
846  //                  var linksBeforeTest = links.Count();
847  //
848  //                  long linksToCreate = 64 * 1024 * 1024 /
     ↪  UInt64UnitedMemoryLinks.LinkSizeInBytes;
849  //
850  //                  ConsoleHelpers.Debug("Creating {0} links.", linksToCreate);
851  //
852  //                  var elapsedTime = Performance.Measure(() =>
853  //                  {
854  //                      for (long i = 0; i < linksToCreate; i++)
855  //                      {
856  //                          links.Create();
857  //                      }
858  //                  });
859  //
860  //                  var linksCreated = links.Count() - linksBeforeTest;
861  //                  var linksPerSecond = linksCreated / elapsedTime.TotalSeconds;
862  //
863  //                  ConsoleHelpers.Debug("Current links count: {0}.", links.Count());
```

```
864  //
865  //                      ConsoleHelpers.Debug("{0} links created in {1} ({2} links per second)",
     ↪   linksCreated, elapsedTime,
866  //                          (long)linksPerSecond);
867  //              }
868  //          }
869  //
870  //          [Fact(Skip = "performance test")]
871  //          public static void Create64BillionLinksInParallel()
872  //          {
873  //              using (var scope = new TempLinksTestScope())
874  //              {
875  //                  var links = scope.Links;
876  //                  var linksBeforeTest = links.Count();
877  //
878  //                  var sw = Stopwatch.StartNew();
879  //
880  //                  long linksToCreate = 64 * 1024 * 1024 /
     ↪   UInt64UnitedMemoryLinks.LinkSizeInBytes;
881  //
882  //                  ConsoleHelpers.Debug("Creating {0} links in parallel.", linksToCreate);
883  //
884  //                  Parallel.For(0, linksToCreate, x => links.Create());
885  //
886  //                  var elapsedTime = sw.Elapsed;
887  //
888  //                  var linksCreated = links.Count() - linksBeforeTest;
889  //                  var linksPerSecond = linksCreated / elapsedTime.TotalSeconds;
890  //
891  //                  ConsoleHelpers.Debug("{0} links created in {1} ({2} links per second)",
     ↪   linksCreated, elapsedTime,
892  //                          (long)linksPerSecond);
893  //              }
894  //          }
895  //
896  //          [Fact(Skip = "useless: O(0), was dependent on creation tests")]
897  //          public static void TestDeletionOfAllLinks()
898  //          {
899  //              using (var scope = new TempLinksTestScope())
900  //              {
901  //                  var links = scope.Links;
902  //                  var linksBeforeTest = links.Count();
903  //
904  //                  ConsoleHelpers.Debug("Deleting all links");
905  //
906  //                  var elapsedTime = Performance.Measure(links.DeleteAll);
907  //
908  //                  var linksDeleted = linksBeforeTest - links.Count();
909  //                  var linksPerSecond = linksDeleted / elapsedTime.TotalSeconds;
910  //
911  //                  ConsoleHelpers.Debug("{0} links deleted in {1} ({2} links per second)",
     ↪   linksDeleted, elapsedTime,
912  //                          (long)linksPerSecond);
913  //              }
914  //          }
915  //
916  //          #endregion
917  //      }
918  // }
```

1.72 ./csharp/Platform.Data.Doublets.Sequences.Tests/Uint64LinksExtensionsTests.cs

```
1   using Platform.Data.Doublets.Memory;
2   using Platform.Data.Doublets.Memory.United.Generic;
3   using Platform.Data.Numbers.Raw;
4   using Platform.Memory;
5   using Platform.Numbers;
6   using Xunit;
7   using Xunit.Abstractions;
8   using TLinkAddress = System.UInt64;
9
10  namespace Platform.Data.Doublets.Sequences.Tests
11  {
12      public class Uint64LinksExtensionsTests
13      {
14          public static ILinks<TLinkAddress> CreateLinks() => CreateLinks<TLinkAddress>(new
              ↪   Platform.IO.TemporaryFile());
15
16          public static ILinks<TLinkAddress> CreateLinks<TLinkAddress>(string dataDBFilename)
17          {
```

```
18          var linksConstants = new
      ↪     LinksConstants<TLinkAddress>(enableExternalReferencesSupport: true);
19          return new UnitedMemoryLinks<TLinkAddress>(new
      ↪     FileMappedResizableDirectMemory(dataDBFilename),
      ↪     UnitedMemoryLinks<TLinkAddress>.DefaultLinksSizeStep, linksConstants,
      ↪     IndexTreeType.Default);
20        }
21        [Fact]
22        public void FormatStructureWithExternalReferenceTest()
23        {
24            ILinks<TLinkAddress> links = CreateLinks();
25            TLinkAddress zero = default;
26            var one = Arithmetic.Increment(zero);
27            var markerIndex = one;
28            var meaningRoot = links.GetOrCreate(markerIndex, markerIndex);
29            var numberMarker = links.GetOrCreate(meaningRoot, Arithmetic.Increment(ref
      ↪     markerIndex));
30            AddressToRawNumberConverter<TLinkAddress> addressToNumberConverter = new();
31            var numberAddress = addressToNumberConverter.Convert(1);
32            var numberLink = links.GetOrCreate(numberMarker, numberAddress);
33            var linkNotation = links.FormatStructure(numberLink, link => link.IsFullPoint(),
      ↪     true);
34            Assert.Equal("(3:(2:1 2) 18446744073709551615)", linkNotation);
35        }
36    }
37 }
```

1.73 ./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  //     public static class UnaryNumberConvertersTests
8  //     {
9  //         [Fact]
10 //         public static void ConvertersTest()
11 //         {
12 //             using (var scope = new TempLinksTestScope())
13 //             {
14 //                 const int N = 10;
15 //                 var links = scope.Links;
16 //                 var meaningRoot = links.CreatePoint();
17 //                 var one = links.CreateAndUpdate(meaningRoot, links.Constants.Itself);
18 //                 var powerOf2ToUnaryNumberConverter = new
      ↪  PowerOf2ToUnaryNumberConverter<ulong>(links, one);
19 //                 var toUnaryNumberConverter = new AddressToUnaryNumberConverter<ulong>(links,
      ↪  powerOf2ToUnaryNumberConverter);
20 //                 var random = new System.Random(0);
21 //                 ulong[] numbers = new ulong[N];
22 //                 ulong[] unaryNumbers = new ulong[N];
23 //                 for (int i = 0; i < N; i++)
24 //                 {
25 //                     numbers[i] = random.NextUInt64();
26 //                     unaryNumbers[i] = toUnaryNumberConverter.Convert(numbers[i]);
27 //                 }
28 //                 var fromUnaryNumberConverterUsingOrOperation = new
      ↪  UnaryNumberToAddressOrOperationConverter<ulong>(links, powerOf2ToUnaryNumberConverter);
29 //                 var fromUnaryNumberConverterUsingAddOperation = new
      ↪  UnaryNumberToAddressAddOperationConverter<ulong>(links, one);
30 //                 for (int i = 0; i < N; i++)
31 //                 {
32 //                     Assert.Equal(numbers[i],
      ↪  fromUnaryNumberConverterUsingOrOperation.Convert(unaryNumbers[i]));
33 //                     Assert.Equal(numbers[i],
      ↪  fromUnaryNumberConverterUsingAddOperation.Convert(unaryNumbers[i]));
34 //                 }
35 //             }
36 //         }
37 //     }
38 // }
```

1.74 ./csharp/Platform.Data.Doublets.Sequences.Tests/UnicodeConvertersTests.cs

```
1  // using Xunit;
2  // using Platform.Converters;
```

```csharp
// 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
// {
//     public static class UnicodeConvertersTests
//     {
//         [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);
//             }
//         }
//
//         [Fact]
//         public static void CharAndRawNumberUnicodeSymbolConvertersTest()
//         {
//             using (var scope = new Scope<Types<HeapResizableDirectMemory,
//     UnitedMemoryLinks<ulong>>>())
//             {
//                 var links = scope.Use<ILinks<ulong>>();
//                 var meaningRoot = links.CreatePoint();
//                 var addressToRawNumberConverter = new AddressToRawNumberConverter<ulong>();
//                 var rawNumberToAddressConverter = new RawNumberToAddressConverter<ulong>();
//                 TestCharAndUnicodeSymbolConverters(links, meaningRoot,
//     addressToRawNumberConverter, rawNumberToAddressConverter);
//             }
//         }
//         private static void TestCharAndUnicodeSymbolConverters(ILinks<ulong> links, ulong
//     meaningRoot, IConverter<ulong> addressToNumberConverter, IConverter<ulong>
//     numberToAddressConverter)
//         {
//             var unicodeSymbolMarker = links.CreateAndUpdate(meaningRoot,
//     links.Constants.Itself);
//             var charToUnicodeSymbolConverter = new CharToUnicodeSymbolConverter<ulong>(links,
//     addressToNumberConverter, unicodeSymbolMarker);
//             var originalCharacter = 'H';
//             var characterLink = charToUnicodeSymbolConverter.Convert(originalCharacter);
//             var unicodeSymbolCriterionMatcher = new TargetMatcher<ulong>(links,
//     unicodeSymbolMarker);
//             var unicodeSymbolToCharConverter = new UnicodeSymbolToCharConverter<ulong>(links,
//     numberToAddressConverter, unicodeSymbolCriterionMatcher);
//             var resultingCharacter = unicodeSymbolToCharConverter.Convert(characterLink);
//             Assert.Equal(originalCharacter, resultingCharacter);
//         }
//
//         [Fact]
//         public static void StringAndUnicodeSequenceConvertersTest()
//         {
//             using (var scope = new TempLinksTestScope())
//             {
//                 var links = scope.Links;
//
//                 var itself = links.Constants.Itself;
```

```
// 
//                 var meaningRoot = links.CreatePoint();
//                 var unaryOne = links.CreateAndUpdate(meaningRoot, itself);
//                 var unicodeSymbolMarker = links.CreateAndUpdate(meaningRoot, itself);
//                 var unicodeSequenceMarker = links.CreateAndUpdate(meaningRoot, itself);
//                 var frequencyMarker = links.CreateAndUpdate(meaningRoot, itself);
//                 var frequencyPropertyMarker = links.CreateAndUpdate(meaningRoot, itself);
// 
//                 var powerOf2ToUnaryNumberConverter = new
//   PowerOf2ToUnaryNumberConverter<ulong>(links, unaryOne);
//                 var addressToUnaryNumberConverter = new
//   AddressToUnaryNumberConverter<ulong>(links, powerOf2ToUnaryNumberConverter);
//                 var charToUnicodeSymbolConverter = new
//   CharToUnicodeSymbolConverter<ulong>(links, addressToUnaryNumberConverter,
//   unicodeSymbolMarker);
// 
//                 var unaryNumberToAddressConverter = new
//   UnaryNumberToAddressOrOperationConverter<ulong>(links, powerOf2ToUnaryNumberConverter);
//                 var unaryNumberIncrementer = new UnaryNumberIncrementer<ulong>(links,
//   unaryOne);
//                 var frequencyIncrementer = new FrequencyIncrementer<ulong>(links,
//   frequencyMarker, unaryOne, unaryNumberIncrementer);
//                 var frequencyPropertyOperator = new PropertyOperator<ulong>(links,
//   frequencyPropertyMarker, frequencyMarker);
//                 var index = new FrequencyIncrementingSequenceIndex<ulong>(links,
//   frequencyPropertyOperator, frequencyIncrementer);
//                 var linkToItsFrequencyNumberConverter = new
//   LinkToItsFrequencyNumberConveter<ulong>(links, frequencyPropertyOperator,
//   unaryNumberToAddressConverter);
//                 var sequenceToItsLocalElementLevelsConverter = new
//   SequenceToItsLocalElementLevelsConverter<ulong>(links, linkToItsFrequencyNumberConverter);
//                 var optimalVariantConverter = new OptimalVariantConverter<ulong>(links,
//   sequenceToItsLocalElementLevelsConverter);
// 
//                 var stringToUnicodeSequenceConverter = new
//   StringToUnicodeSequenceConverter<ulong>(links, charToUnicodeSymbolConverter, index,
//   optimalVariantConverter, unicodeSequenceMarker);
// 
//                 var originalString = "Hello";
// 
//                 var unicodeSequenceLink =
//   stringToUnicodeSequenceConverter.Convert(originalString);
// 
//                 var unicodeSymbolCriterionMatcher = new TargetMatcher<ulong>(links,
//   unicodeSymbolMarker);
//                 var unicodeSymbolToCharConverter = new
//   UnicodeSymbolToCharConverter<ulong>(links, unaryNumberToAddressConverter,
//   unicodeSymbolCriterionMatcher);
// 
//                 var unicodeSequenceCriterionMatcher = new TargetMatcher<ulong>(links,
//   unicodeSequenceMarker);
// 
//                 var sequenceWalker = new LeveledSequenceWalker<ulong>(links,
//   unicodeSymbolCriterionMatcher.IsMatched);
// 
//                 var unicodeSequenceToStringConverter = new
//   UnicodeSequenceToStringConverter<ulong>(links, unicodeSequenceCriterionMatcher,
//   sequenceWalker, unicodeSymbolToCharConverter);
// 
//                 var resultingString =
//   unicodeSequenceToStringConverter.Convert(unicodeSequenceLink);
// 
//                 Assert.Equal(originalString, resultingString);
//             }
//         }
//     }
// }
```

Index