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

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

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

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

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

```
            var loopedLength = length - (length % 2);
            for (var i = 0; i < loopedLength; i += 2)
            {
                destination[i / 2] = _links.GetOrCreate(source[i], source[i + 1]);
            }
            if (length > loopedLength)
            {
                destination[length / 2] = source[length - 1];
            }
        }
    }
}
```

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

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

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

namespace Platform.Data.Doublets.Sequences.Converters
{
    /// <remarks>
    /// TODO: Возможно будет лучше если алгоритм будет выполняться полностью изолированно от Links на этапе сжатия.
    ///     А именно будет создаваться временный список пар необходимых для выполнения сжатия, в таком случае тип значения элемента массива может быть любым, как char так и ulong.
    ///     Как только список/словарь пар был выявлен можно разом выполнить создание всех этих пар, а так же разом выполнить замену.
    /// </remarks>
    public class CompressingConverter<TLink> : LinksListToSequenceConverterBase<TLink>
    {
        private static readonly LinksConstants<TLink> _constants = Default<LinksConstants<TLink>>.Instance;
        private static readonly EqualityComparer<TLink> _equalityComparer = EqualityComparer<TLink>.Default;
        private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
        private static readonly TLink _zero = default;
        private static readonly TLink _one = Arithmetic.Increment(_zero);
        private readonly IConverter<IList<TLink>, TLink> _baseConverter;
        private readonly LinkFrequenciesCache<TLink> _doubletFrequenciesCache;
        private readonly TLink _minFrequencyToCompress;
        private readonly bool _doInitialFrequenciesIncrement;
        private Doublet<TLink> _maxDoublet;
        private LinkFrequency<TLink> _maxDoubletData;
        private struct HalfDoublet
        {
            /// <summary>
            /// <para>
            /// The element.
            /// </para>
            /// <para></para>
            /// </summary>
            public TLink Element;
            /// <summary>
            /// <para>
            /// The doublet data.
            /// </para>
            /// <para></para>
            /// </summary>
            public LinkFrequency<TLink> DoubletData;

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

```
            /// </param>
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public HalfDoublet(TLink element, LinkFrequency<TLink> doubletData)
            {
                Element = element;
                DoubletData = doubletData;
            }

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

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

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

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

```
136         /// </summary>
137         /// <param name="links">
138         /// <para>A links.</para>
139         /// <para></para>
140         /// </param>
141         /// <param name="baseConverter">
142         /// <para>A base converter.</para>
143         /// <para></para>
144         /// </param>
145         /// <param name="doubletFrequenciesCache">
146         /// <para>A doublet frequencies cache.</para>
147         /// <para></para>
148         /// </param>
149         /// <param name="minFrequencyToCompress">
150         /// <para>A min frequency to compress.</para>
151         /// <para></para>
152         /// </param>
153         /// <param name="doInitialFrequenciesIncrement">
154         /// <para>A do initial frequencies increment.</para>
155         /// <para></para>
156         /// </param>
157         [MethodImpl(MethodImplOptions.AggressiveInlining)]
158         public CompressingConverter(ILinks<TLink> links, IConverter<IList<TLink>, TLink>
            ↪ baseConverter, LinkFrequenciesCache<TLink> doubletFrequenciesCache, TLink
            ↪ minFrequencyToCompress, bool doInitialFrequenciesIncrement)
159             : base(links)
160         {
161             _baseConverter = baseConverter;
162             _doubletFrequenciesCache = doubletFrequenciesCache;
163             if (_comparer.Compare(minFrequencyToCompress, _one) < 0)
164             {
165                 minFrequencyToCompress = _one;
166             }
167             _minFrequencyToCompress = minFrequencyToCompress;
168             _doInitialFrequenciesIncrement = doInitialFrequenciesIncrement;
169             ResetMaxDoublet();
170         }
171 
172         /// <summary>
173         /// <para>
174         /// Converts the source.
175         /// </para>
176         /// <para></para>
177         /// </summary>
178         /// <param name="source">
179         /// <para>The source.</para>
180         /// <para></para>
181         /// </param>
182         /// <returns>
183         /// <para>The link</para>
184         /// <para></para>
185         /// </returns>
186         [MethodImpl(MethodImplOptions.AggressiveInlining)]
187         public override TLink Convert(IList<TLink> source) =>
            ↪ _baseConverter.Convert(Compress(source));
188         [MethodImpl(MethodImplOptions.AggressiveInlining)]
189         private IList<TLink> Compress(IList<TLink> sequence)
190         {
191             if (sequence.IsNullOrEmpty())
192             {
193                 return null;
194             }
195             if (sequence.Count == 1)
196             {
197                 return sequence;
198             }
199             if (sequence.Count == 2)
200             {
201                 return new[] { _links.GetOrCreate(sequence[0], sequence[1]) };
202             }
203             // TODO: arraypool with min size (to improve cache locality) or stackallow with Sigil
204             var copy = new HalfDoublet[sequence.Count];
205             Doublet<TLink> doublet = default;
206             for (var i = 1; i < sequence.Count; i++)
207             {
208                 doublet = new Doublet<TLink>(sequence[i - 1], sequence[i]);
209                 LinkFrequency<TLink> data;
210                 if (_doInitialFrequenciesIncrement)
```

```
                {
                    data = _doubletFrequenciesCache.IncrementFrequency(ref doublet);
                }
                else
                {
                    data = _doubletFrequenciesCache.GetFrequency(ref doublet);
                    if (data == null)
                    {
                        throw new NotSupportedException("If you ask not to increment
                        ↪  frequencies, it is expected that all frequencies for the sequence
                        ↪  are prepared.");
                    }
                }
                copy[i - 1].Element = sequence[i - 1];
                copy[i - 1].DoubletData = data;
                UpdateMaxDoublet(ref doublet, data);
            }
            copy[sequence.Count - 1].Element = sequence[sequence.Count - 1];
            copy[sequence.Count - 1].DoubletData = new LinkFrequency<TLink>();
            if (_comparer.Compare(_maxDoubletData.Frequency, default) > 0)
            {
                var newLength = ReplaceDoublets(copy);
                sequence = new TLink[newLength];
                for (int i = 0; i < newLength; i++)
                {
                    sequence[i] = copy[i].Element;
                }
            }
            return sequence;
        }
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private int ReplaceDoublets(HalfDoublet[] copy)
        {
            var oldLength = copy.Length;
            var newLength = copy.Length;
            while (_comparer.Compare(_maxDoubletData.Frequency, default) > 0)
            {
                var maxDoubletSource = _maxDoublet.Source;
                var maxDoubletTarget = _maxDoublet.Target;
                if (_equalityComparer.Equals(_maxDoubletData.Link, _constants.Null))
                {
                    _maxDoubletData.Link = _links.GetOrCreate(maxDoubletSource,
                    ↪  maxDoubletTarget);
                }
                var maxDoubletReplacementLink = _maxDoubletData.Link;
                oldLength--;
                var oldLengthMinusTwo = oldLength - 1;
                // Substitute all usages
                int w = 0, r = 0; // (r == read, w == write)
                for (; r < oldLength; r++)
                {
                    if (_equalityComparer.Equals(copy[r].Element, maxDoubletSource) &&
                    ↪  _equalityComparer.Equals(copy[r + 1].Element, maxDoubletTarget))
                    {
                        if (r > 0)
                        {
                            var previous = copy[w - 1].Element;
                            copy[w - 1].DoubletData.DecrementFrequency();
                            copy[w - 1].DoubletData =
                            ↪  _doubletFrequenciesCache.IncrementFrequency(previous,
                            ↪  maxDoubletReplacementLink);
                        }
                        if (r < oldLengthMinusTwo)
                        {
                            var next = copy[r + 2].Element;
                            copy[r + 1].DoubletData.DecrementFrequency();
                            copy[w].DoubletData = _doubletFrequenciesCache.IncrementFrequency(ma
                            ↪  xDoubletReplacementLink,
                            ↪  next);
                        }
                        copy[w++].Element = maxDoubletReplacementLink;
                        r++;
                        newLength--;
                    }
                    else
                    {
                        copy[w++] = copy[r];
                    }
```

```
                }
                if (w < newLength)
                {
                    copy[w] = copy[r];
                }
                oldLength = newLength;
                ResetMaxDoublet();
                UpdateMaxDoublet(copy, newLength);
            }
            return newLength;
        }
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private void ResetMaxDoublet()
        {
            _maxDoublet = new Doublet<TLink>();
            _maxDoubletData = new LinkFrequency<TLink>();
        }
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private void UpdateMaxDoublet(HalfDoublet[] copy, int length)
        {
            Doublet<TLink> doublet = default;
            for (var i = 1; i < length; i++)
            {
                doublet = new Doublet<TLink>(copy[i - 1].Element, copy[i].Element);
                UpdateMaxDoublet(ref doublet, copy[i - 1].DoubletData);
            }
        }
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private void UpdateMaxDoublet(ref Doublet<TLink> doublet, LinkFrequency<TLink> 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

```
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{TLink}"/>
    /// <seealso cref="IConverter{IList{TLink}, TLink}"/>
    public abstract class LinksListToSequenceConverterBase<TLink> : LinksOperatorBase<TLink>,
    ↪  IConverter<IList<TLink>, TLink>
    {
        /// <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<TLink> 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 TLink Convert(IList<TLink> source);
    }
}
```

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

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

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

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

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

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

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

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

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

```
                                GetGreatestNeigbourLowerThanCurrentOrCurrent(previousLevel,
                                ↪  currentLevel, levels[i + 1]);
                            levels[w] = newLevel;
                            previousLevel = currentLevel;
                            w++;
                            levelRepeat = 0;
                            skipOnce = true;
                        }
                        else if (i == length - 1)
                        {
                            sequence[w] = sequence[i];
                            levels[w] = levels[i];
                            w++;
                        }
                    }
                    else
                    {
                        currentLevel = levels[i];
                        levelRepeat = 1;
                        if (skipOnce)
                        {
                            skipOnce = false;
                        }
                        else
                        {
                            sequence[w] = sequence[i - 1];
                            levels[w] = levels[i - 1];
                            previousLevel = levels[w];
                            w++;
                        }
                        if (i == length - 1)
                        {
                            sequence[w] = sequence[i];
                            levels[w] = levels[i];
                            w++;
                        }
                    }
                }
                length = w;
            }
            return _links.GetOrCreate(sequence[0], sequence[1]);
        }
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static TLink GetGreatestNeigbourLowerThanCurrentOrCurrent(TLink previous, TLink
        ↪  current, TLink next)
        {
            return _comparer.Compare(previous, next) > 0
                ? _comparer.Compare(previous, current) < 0 ? previous : current
                : _comparer.Compare(next, current) < 0 ? next : current;
        }
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static TLink GetNextLowerThanCurrentOrCurrent(TLink current, TLink next) =>
        ↪  _comparer.Compare(next, current) < 0 ? next : current;
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static TLink GetPreviousLowerThanCurrentOrCurrent(TLink previous, TLink current)
        ↪  => _comparer.Compare(previous, current) < 0 ? previous : current;
    }
}
```

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

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

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

namespace Platform.Data.Doublets.Sequences.Converters
{
    /// <summary>
    /// <para>
    /// Represents the sequence to its local element levels converter.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="LinksOperatorBase{TLink}"/>
    /// <seealso cref="IConverter{IList{TLink}}"/>
    public class SequenceToItsLocalElementLevelsConverter<TLink> : LinksOperatorBase<TLink>,
    ↪  IConverter<IList<TLink>>
    {
```

```
19          private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
20          private readonly IConverter<Doublet<TLink>, TLink> _linkToItsFrequencyToNumberConveter;
21
22          /// <summary>
23          /// <para>
24          /// Initializes a new <see cref="SequenceToItsLocalElementLevelsConverter"/> 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="linkToItsFrequencyToNumberConveter">
33          /// <para>A link to its frequency to number conveter.</para>
34          /// <para></para>
35          /// </param>
36          [MethodImpl(MethodImplOptions.AggressiveInlining)]
37          public SequenceToItsLocalElementLevelsConverter(ILinks<TLink> links,
            ↪  IConverter<Doublet<TLink>, TLink> linkToItsFrequencyToNumberConveter) : base(links)
            ↪  => _linkToItsFrequencyToNumberConveter = linkToItsFrequencyToNumberConveter;
38
39          /// <summary>
40          /// <para>
41          /// Converts the sequence.
42          /// </para>
43          /// <para></para>
44          /// </summary>
45          /// <param name="sequence">
46          /// <para>The sequence.</para>
47          /// <para></para>
48          /// </param>
49          /// <returns>
50          /// <para>The levels.</para>
51          /// <para></para>
52          /// </returns>
53          [MethodImpl(MethodImplOptions.AggressiveInlining)]
54          public IList<TLink> Convert(IList<TLink> sequence)
55          {
56              var levels = new TLink[sequence.Count];
57              levels[0] = GetFrequencyNumber(sequence[0], sequence[1]);
58              for (var i = 1; i < sequence.Count - 1; i++)
59              {
60                  var previous = GetFrequencyNumber(sequence[i - 1], sequence[i]);
61                  var next = GetFrequencyNumber(sequence[i], sequence[i + 1]);
62                  levels[i] = _comparer.Compare(previous, next) > 0 ? previous : next;
63              }
64              levels[levels.Length - 1] = GetFrequencyNumber(sequence[sequence.Count - 2],
                ↪  sequence[sequence.Count - 1]);
65              return levels;
66          }
67
68          /// <summary>
69          /// <para>
70          /// Gets the frequency number using the specified source.
71          /// </para>
72          /// <para></para>
73          /// </summary>
74          /// <param name="source">
75          /// <para>The source.</para>
76          /// <para></para>
77          /// </param>
78          /// <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 TLink GetFrequencyNumber(TLink source, TLink target) =>
            ↪  _linkToItsFrequencyToNumberConveter.Convert(new Doublet<TLink>(source, target));
88      }
89  }
```

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

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

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

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

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

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

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

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

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

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

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

        /// <summary>
        /// <para>
        /// Determines whether this instance is matched.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="sequenceCandidate">
        /// <para>The sequence candidate.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public bool IsMatched(TLink sequenceCandidate)
            => _equalityComparer.Equals(_links.GetSource(sequenceCandidate), _sequenceMarkerLink)
            || !_equalityComparer.Equals(_links.SearchOrDefault(_sequenceMarkerLink,
            ↪  sequenceCandidate), _links.Constants.Null);
    }
}
```

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

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

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

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

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

```
39          /// <param name="heightProvider">
40          /// <para>A height provider.</para>
41          /// <para></para>
42          /// </param>
43          [MethodImpl(MethodImplOptions.AggressiveInlining)]
44          public DefaultSequenceAppender(ILinks<TLink> links, IStack<TLink> stack,
            ↪  ISequenceHeightProvider<TLink> 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 TLink Append(TLink sequence, TLink 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  }
```

1.9 ./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<TLink> : ICounter<int>
18      {
19          private readonly IProvider<IList<KeyValuePair<IList<TLink>, IList<TLink>>>>
            ↪  _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<TLink>,
            ↪  IList<TLink>>>> 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  }
```

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

```
1   using System;
2   using System.Linq;
3   using System.Collections.Generic;
4   using System.Runtime.CompilerServices;
5   using Platform.Interfaces;
6   using Platform.Collections;
7   using Platform.Collections.Lists;
8   using Platform.Collections.Segments;
9   using Platform.Collections.Segments.Walkers;
10  using Platform.Singletons;
11  using Platform.Converters;
12  using Platform.Data.Doublets.Unicode;
13
14  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
15
16  namespace Platform.Data.Doublets.Sequences
17  {
18      /// <summary>
19      /// <para>
20      /// Represents the duplicate segments provider.
21      /// </para>
22      /// <para></para>
23      /// </summary>
24      /// <seealso cref="DictionaryBasedDuplicateSegmentsWalkerBase{TLink}"/>
25      /// <seealso cref="IProvider{IList{KeyValuePair{IList{TLink}, IList{TLink}}}}"/>
26      public class DuplicateSegmentsProvider<TLink> :
        ↪  DictionaryBasedDuplicateSegmentsWalkerBase<TLink>,
        ↪  IProvider<IList<KeyValuePair<IList<TLink>, IList<TLink>>>>
27      {
28          private static readonly UncheckedConverter<TLink, long> _addressToInt64Converter =
            ↪  UncheckedConverter<TLink, long>.Default;
29          private static readonly UncheckedConverter<TLink, ulong> _addressToUInt64Converter =
            ↪  UncheckedConverter<TLink, ulong>.Default;
30          private static readonly UncheckedConverter<ulong, TLink> _uInt64ToAddressConverter =
            ↪  UncheckedConverter<ulong, TLink>.Default;
31          private readonly ILinks<TLink> _links;
32          private readonly ILinks<TLink> _sequences;
33          private HashSet<KeyValuePair<IList<TLink>, IList<TLink>>> _groups;
34          private BitString _visited;
35          private class ItemEquilityComparer : IEqualityComparer<KeyValuePair<IList<TLink>,
            ↪  IList<TLink>>>
36          {
```

```
37              private readonly IListEqualityComparer<TLink> _listComparer;
38
39              /// <summary>
40              /// <para>
41              /// Initializes a new <see cref="ItemEquilityComparer"/> instance.
42              /// </para>
43              /// <para></para>
44              /// </summary>
45              public ItemEquilityComparer() => _listComparer =
                ↪  Default<IListEqualityComparer<TLink>>.Instance;
46
47              /// <summary>
48              /// <para>
49              /// Determines whether this instance equals.
50              /// </para>
51              /// <para></para>
52              /// </summary>
53              /// <param name="left">
54              /// <para>The left.</para>
55              /// <para></para>
56              /// </param>
57              /// <param name="right">
58              /// <para>The right.</para>
59              /// <para></para>
60              /// </param>
61              /// <returns>
62              /// <para>The bool</para>
63              /// <para></para>
64              /// </returns>
65              [MethodImpl(MethodImplOptions.AggressiveInlining)]
66              public bool Equals(KeyValuePair<IList<TLink>, IList<TLink>> left,
                ↪  KeyValuePair<IList<TLink>, IList<TLink>> right) =>
                ↪  _listComparer.Equals(left.Key, right.Key) && _listComparer.Equals(left.Value,
                ↪  right.Value);
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<TLink>, IList<TLink>> pair) =>
                ↪  (_listComparer.GetHashCode(pair.Key),
                ↪  _listComparer.GetHashCode(pair.Value)).GetHashCode();
84          }
85          private class ItemComparer : IComparer<KeyValuePair<IList<TLink>, IList<TLink>>>
86          {
87              private readonly IListComparer<TLink> _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<TLink>>.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">
```

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

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

        /// <summary>
        /// <para>
        /// Gets this instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <returns>
        /// <para>The result list.</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public IList<KeyValuePair<IList<TLink>, IList<TLink>>> Get()
        {
            _groups = new HashSet<KeyValuePair<IList<TLink>,
            ↪  IList<TLink>>>(Default<ItemEquilityComparer>.Instance);
            var links = _links;
            var count = links.Count();
            _visited = new BitString(_addressToInt64Converter.Convert(count) + 1L);
            links.Each(link =>
            {
                var linkIndex = links.GetIndex(link);
                var linkBitIndex = _addressToInt64Converter.Convert(linkIndex);
                var constants = links.Constants;
                if (!_visited.Get(linkBitIndex))
                {
                    var sequenceElements = new List<TLink>();
                    var filler = new ListFiller<TLink, TLink>(sequenceElements, constants.Break);
                    _sequences.Each(filler.AddSkipFirstAndReturnConstant, new
                    ↪  LinkAddress<TLink>(linkIndex));
                    if (sequenceElements.Count > 2)
                    {
                        WalkAll(sequenceElements);
                    }
                }
                return constants.Continue;
            });
```

```
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<TLink> CreateSegment(IList<TLink> elements, int offset, int
            ↪  length) => new Segment<TLink>(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<TLink> segment)
233         {
234             var duplicates = CollectDuplicatesForSegment(segment);
235             if (duplicates.Count > 1)
236             {
237                 _groups.Add(new KeyValuePair<IList<TLink>, IList<TLink>>(segment.ToArray(),
                    ↪  duplicates));
238             }
239         }
240         [MethodImpl(MethodImplOptions.AggressiveInlining)]
241         private List<TLink> CollectDuplicatesForSegment(Segment<TLink> segment)
242         {
243             var duplicates = new List<TLink>();
244             var readAsElement = new HashSet<TLink>();
245             var restrictions = segment.ShiftRight();
246             var constants = _links.Constants;
247             restrictions[0] = constants.Any;
248             _sequences.Each(sequence =>
249             {
250                 var sequenceIndex = sequence[constants.IndexPart];
251                 duplicates.Add(sequenceIndex);
252                 readAsElement.Add(sequenceIndex);
253                 return constants.Continue;
254             }, restrictions);
255             if (duplicates.Any(x => _visited.Get(_addressToInt64Converter.Convert(x))))
256             {
257                 return new List<TLink>();
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((H
                    ↪ ashSet<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<TLink>, IList<TLink>> 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 }
```

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

```
1  using System;
2  using System.Collections.Generic;
3  using System.Runtime.CompilerServices;
4  using Platform.Interfaces;
5  using Platform.Numbers;
6
7  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
8
9  namespace Platform.Data.Doublets.Sequences.Frequencies.Cache
10 {
11     /// <remarks>
12     /// Can be used to operate with many CompressingConverters (to keep global frequencies data
       ↪ between them).
13     /// TODO: Extract interface to implement frequencies storage inside Links storage
14     /// </remarks>
15     public class LinkFrequenciesCache<TLink> : LinksOperatorBase<TLink>
16     {
17         private static readonly EqualityComparer<TLink> _equalityComparer =
           ↪ EqualityComparer<TLink>.Default;
18         private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
19         private static readonly TLink _zero = default;
20         private static readonly TLink _one = Arithmetic.Increment(_zero);
21         private readonly Dictionary<Doublet<TLink>, LinkFrequency<TLink>> _doubletsCache;
22         private readonly ICounter<TLink, TLink> _frequencyCounter;
23
24         /// <summary>
25         /// <para>
26         /// Initializes a new <see cref="LinkFrequenciesCache"/> instance.
27         /// </para>
28         /// <para></para>
29         /// </summary>
```

```
        /// <param name="links">
        /// <para>A links.</para>
        /// <para></para>
        /// </param>
        /// <param name="frequencyCounter">
        /// <para>A frequency counter.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public LinkFrequenciesCache(ILinks<TLink> links, ICounter<TLink, TLink> frequencyCounter)
            : base(links)
        {
            _doubletsCache = new Dictionary<Doublet<TLink>, LinkFrequency<TLink>>(4096,
            ↪  DoubletComparer<TLink>.Default);
            _frequencyCounter = frequencyCounter;
        }

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

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

```
                IncrementFrequency(sequence[i - 1], sequence[i]);
            }
        }

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

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

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

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

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

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

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

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

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

```
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<TLink>
15     {
16         /// <summary>
17         /// <para>
18         /// Gets or sets the frequency value.
19         /// </para>
20         /// <para></para>
21         /// </summary>
22         public TLink Frequency { get; set; }
23         /// <summary>
24         /// <para>
25         /// Gets or sets the link value.
26         /// </para>
27         /// <para></para>
28         /// </summary>
29         public TLink 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(TLink frequency, TLink 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<TLink>.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<TLink>.Decrement(Frequency);
78
79         /// <summary>
80         /// <para>
```

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

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

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

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

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

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

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

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

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

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

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

```
    /// Represents the marked sequence symbol frequency one off counter.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="SequenceSymbolFrequencyOneOffCounter{TLink}"/>
    public class MarkedSequenceSymbolFrequencyOneOffCounter<TLink> :
    ↪ SequenceSymbolFrequencyOneOffCounter<TLink>
    {
        private readonly ICriterionMatcher<TLink> _markedSequenceMatcher;

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

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

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

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

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

namespace Platform.Data.Doublets.Sequences.Frequencies.Counters
{
    /// <summary>
    /// <para>
    /// Represents the sequence symbol frequency one off counter.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="ICounter{TLink}"/>
    public class SequenceSymbolFrequencyOneOffCounter<TLink> : ICounter<TLink>
```

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

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

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

        /// <summary>
        /// <para>
        /// Counts this instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <returns>
        /// <para>The total.</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public virtual TLink Count()
        {
            if (_comparer.Compare(_total, default) > 0)
            {
                return _total;
            }
```

```
            StopableSequenceWalker.WalkRight(_sequenceLink, _links.GetSource, _links.GetTarget,
            ↪  IsElement, VisitElement);
            return _total;
        }
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private bool IsElement(TLink x) => _equalityComparer.Equals(x, _symbol) ||
        ↪  _links.IsPartialPoint(x); // TODO: Use SequenceElementCreteriaMatcher instead of
        ↪  IsPartialPoint
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private bool VisitElement(TLink element)
        {
            if (_equalityComparer.Equals(element, _symbol))
            {
                _total = Arithmetic.Increment(_total);
            }
            return true;
        }
    }
}
```

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

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

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

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

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

        /// <summary>
        /// <para>
        /// Counts the argument.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="argument">
        /// <para>The argument.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
56          public TLink Count(TLink argument) => new
            ↪  TotalMarkedSequenceSymbolFrequencyOneOffCounter<TLink>(_links,
            ↪  _markedSequenceMatcher, argument).Count();
57      }
58  }
```

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

```
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{TLink}"/>
16      public class TotalMarkedSequenceSymbolFrequencyOneOffCounter<TLink> :
        ↪  TotalSequenceSymbolFrequencyOneOffCounter<TLink>
17      {
18          private readonly ICriterionMatcher<TLink> _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<TLink> links,
            ↪  ICriterionMatcher<TLink> markedSequenceMatcher, TLink 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(TLink link)
55          {
56              var symbolFrequencyCounter = new
                ↪  MarkedSequenceSymbolFrequencyOneOffCounter<TLink>(_links,
                ↪  _markedSequenceMatcher, link, _symbol);
57              _total = Arithmetic.Add(_total, symbolFrequencyCounter.Count());
58          }
59      }
60  }
```

1.18 ./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
```

```

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

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

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

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

```

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

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

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

        /// <summary>
        /// <para>
        /// The links.
        /// </para>
        /// <para></para>
        /// </summary>
        protected readonly ILinks<TLink> _links;
        /// <summary>
        /// <para>
```

```
        /// The symbol.
        /// </para>
        /// <para></para>
        /// </summary>
        protected readonly TLink _symbol;
        /// <summary>
        /// <para>
        /// The visits.
        /// </para>
        /// <para></para>
        /// </summary>
        protected readonly HashSet<TLink> _visits;
        /// <summary>
        /// <para>
        /// The total.
        /// </para>
        /// <para></para>
        /// </summary>
        protected TLink _total;

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

        /// <summary>
        /// <para>
        /// Counts this instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <returns>
        /// <para>The total.</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TLink Count()
        {
            if (_comparer.Compare(_total, default) > 0 || _visits.Count > 0)
            {
                return _total;
            }
            CountCore(_symbol);
            return _total;
        }
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private void CountCore(TLink 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(TLink link)
        {
            var symbolFrequencyCounter = new SequenceSymbolFrequencyOneOffCounter<TLink>(_links,
            ↪  link, _symbol);
            _total = Arithmetic.Add(_total, symbolFrequencyCounter.Count());
        }
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private TLink EachElementHandler(IList<TLink> doublet)
        {
            var constants = _links.Constants;
            var doubletIndex = doublet[constants.IndexPart];
            if (_visits.Add(doubletIndex))
            {
                CountCore(doubletIndex);
            }
            return constants.Continue;
        }
    }
}
```

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

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

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

namespace Platform.Data.Doublets.Sequences.HeightProviders
{
    /// <summary>
    /// <para>
    /// Represents the cached sequence height provider.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="ISequenceHeightProvider{TLink}"/>
    public class CachedSequenceHeightProvider<TLink> : ISequenceHeightProvider<TLink>
    {
        private static readonly EqualityComparer<TLink> _equalityComparer =
        ↪  EqualityComparer<TLink>.Default;
        private readonly TLink _heightPropertyMarker;
        private readonly ISequenceHeightProvider<TLink> _baseHeightProvider;
        private readonly IConverter<TLink> _addressToUnaryNumberConverter;
        private readonly IConverter<TLink> _unaryNumberToAddressConverter;
        private readonly IProperties<TLink, TLink, TLink> _propertyOperator;

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="CachedSequenceHeightProvider"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="baseHeightProvider">
        /// <para>A base height provider.</para>
        /// <para></para>
        /// </param>
        /// <param name="addressToUnaryNumberConverter">
        /// <para>A address to unary number converter.</para>
        /// <para></para>
        /// </param>
        /// <param name="unaryNumberToAddressConverter">
        /// <para>A unary number to address converter.</para>
        /// <para></para>
        /// </param>
        /// <param name="heightPropertyMarker">
        /// <para>A height property marker.</para>
        /// <para></para>
        /// </param>
        /// <param name="propertyOperator">
```

```
        /// <para>A property operator.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public CachedSequenceHeightProvider(
            ISequenceHeightProvider<TLink> baseHeightProvider,
            IConverter<TLink> addressToUnaryNumberConverter,
            IConverter<TLink> unaryNumberToAddressConverter,
            TLink heightPropertyMarker,
            IProperties<TLink, TLink, TLink> propertyOperator)
        {
            _heightPropertyMarker = heightPropertyMarker;
            _baseHeightProvider = baseHeightProvider;
            _addressToUnaryNumberConverter = addressToUnaryNumberConverter;
            _unaryNumberToAddressConverter = unaryNumberToAddressConverter;
            _propertyOperator = propertyOperator;
        }

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

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

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

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

namespace Platform.Data.Doublets.Sequences.HeightProviders
{
    /// <summary>
    /// <para>
    /// Represents the default sequence right height provider.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="LinksOperatorBase{TLink}"/>
    /// <seealso cref="ISequenceHeightProvider{TLink}"/>
    public class DefaultSequenceRightHeightProvider<TLink> : LinksOperatorBase<TLink>,
    ↪ ISequenceHeightProvider<TLink>
    {
        private readonly ICriterionMatcher<TLink> _elementMatcher;

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

```
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<TLink> links, ICriterionMatcher<TLink>
            ↪  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 TLink Get(TLink sequence)
54          {
55              var height = default(TLink);
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  }
```

1.22 ./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{TLink, TLink}"/>
14      public interface ISequenceHeightProvider<TLink> : IProvider<TLink, TLink>
15      {
16      }
17  }
```

1.23 ./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{TLink}"/>
16      /// <seealso cref="IIncrementer{TLink}"/>
17      public class FrequencyIncrementer<TLink> : LinksOperatorBase<TLink>, IIncrementer<TLink>
```

```
    {
        private static readonly EqualityComparer<TLink> _equalityComparer = EqualityComparer<TLink>.Default;
        private readonly TLink _frequencyMarker;
        private readonly TLink _unaryOne;
        private readonly IIncrementer<TLink> _unaryNumberIncrementer;

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="FrequencyIncrementer"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="links">
        /// <para>A links.</para>
        /// <para></para>
        /// </param>
        /// <param name="frequencyMarker">
        /// <para>A frequency marker.</para>
        /// <para></para>
        /// </param>
        /// <param name="unaryOne">
        /// <para>A unary one.</para>
        /// <para></para>
        /// </param>
        /// <param name="unaryNumberIncrementer">
        /// <para>A unary number incrementer.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public FrequencyIncrementer(ILinks<TLink> links, TLink frequencyMarker, TLink unaryOne, IIncrementer<TLink> unaryNumberIncrementer)
            : base(links)
        {
            _frequencyMarker = frequencyMarker;
            _unaryOne = unaryOne;
            _unaryNumberIncrementer = unaryNumberIncrementer;
        }

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

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

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

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

namespace Platform.Data.Doublets.Incrementers
{
    /// <summary>
    /// <para>
```

```
    /// Represents the unary number incrementer.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="LinksOperatorBase{TLink}"/>
    /// <seealso cref="IIncrementer{TLink}"/>
    public class UnaryNumberIncrementer<TLink> : LinksOperatorBase<TLink>, IIncrementer<TLink>
    {
        private static readonly EqualityComparer<TLink> _equalityComparer =
        ↪  EqualityComparer<TLink>.Default;
        private readonly TLink _unaryOne;

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

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

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

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

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

namespace Platform.Data.Doublets.Sequences.Indexes
{
    /// <summary>
    /// <para>
    /// Represents the cached frequency incrementing sequence index.
    /// </para>
```

```
13      /// <para></para>
14      /// </summary>
15      /// <seealso cref="ISequenceIndex{TLink}"/>
16      public class CachedFrequencyIncrementingSequenceIndex<TLink> : ISequenceIndex<TLink>
17      {
18          private static readonly EqualityComparer<TLink> _equalityComparer =
            ↪  EqualityComparer<TLink>.Default;
19          private readonly LinkFrequenciesCache<TLink> _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<TLink> 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<TLink> 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(TLink source, TLink 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>
```

```
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public bool MightContain(IList<TLink> sequence)
        {
            var indexed = true;
            var i = sequence.Count;
            while (--i >= 1 && (indexed = IsIndexed(sequence[i - 1], sequence[i]))) { }
            return indexed;
        }
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private bool IsIndexed(TLink source, TLink target)
        {
            var frequency = _cache.GetFrequency(source, target);
            if (frequency == null)
            {
                return false;
            }
            return !_equalityComparer.Equals(frequency.Frequency, default);
        }
    }
}
```

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

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

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

namespace Platform.Data.Doublets.Sequences.Indexes
{
    /// <summary>
    /// <para>
    /// Represents the frequency incrementing sequence index.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="SequenceIndex{TLink}"/>
    /// <seealso cref="ISequenceIndex{TLink}"/>
    public class FrequencyIncrementingSequenceIndex<TLink> : SequenceIndex<TLink>,
    ↪  ISequenceIndex<TLink>
    {
        private static readonly EqualityComparer<TLink> _equalityComparer =
        ↪  EqualityComparer<TLink>.Default;
        private readonly IProperty<TLink, TLink> _frequencyPropertyOperator;
        private readonly IIncrementer<TLink> _frequencyIncrementer;

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

        /// <summary>
        /// <para>
        /// Determines whether this instance add.
```

```
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<TLink> 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(TLink source, TLink 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(TLink link)
89          {
90              var previousFrequency = _frequencyPropertyOperator.Get(link);
91              var frequency = _frequencyIncrementer.Increment(previousFrequency);
92              _frequencyPropertyOperator.Set(link, frequency);
93          }
94      }
95  }
```

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

```
1   using System.Collections.Generic;
2   using System.Runtime.CompilerServices;
3
4   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6   namespace Platform.Data.Doublets.Sequences.Indexes
7   {
8       /// <summary>
9       /// <para>
10      /// Defines the sequence index.
11      /// </para>
12      /// <para></para>
13      /// </summary>
14      public interface ISequenceIndex<TLink>
15      {
16          /// <summary>
17          /// Индексирует последовательность глобально, и возвращает значение,
18          /// определяющие была ли запрошенная последовательность проиндексирована ранее.
19          /// </summary>
20          /// <param name="sequence">Последовательность для индексации.</param>
21          [MethodImpl(MethodImplOptions.AggressiveInlining)]
22          bool Add(IList<TLink> sequence);
23
24          /// <summary>
25          /// <para>
26          /// Determines whether this instance might contain.
27          /// </para>
28          /// <para></para>
29          /// </summary>
30          /// <param name="sequence">
31          /// <para>The sequence.</para>
32          /// <para></para>
33          /// </param>
```

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

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

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

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

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

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

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

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

```
67          /// <para>The sequence.</para>
68          /// <para></para>
69          /// </param>
70          /// <returns>
71          /// <para>The indexed.</para>
72          /// <para></para>
73          /// </returns>
74          [MethodImpl(MethodImplOptions.AggressiveInlining)]
75          public virtual bool MightContain(IList<TLink> sequence)
76          {
77              var indexed = true;
78              var i = sequence.Count;
79              while (--i >= 1 && (indexed =
                ↪  !_equalityComparer.Equals(_links.SearchOrDefault(sequence[i - 1], sequence[i]),
                ↪  default))) { }
80              return indexed;
81          }
82      }
83  }
```

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

```
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 synchronized sequence index.
11      /// </para>
12      /// <para></para>
13      /// </summary>
14      /// <seealso cref="ISequenceIndex{TLink}"/>
15      public class SynchronizedSequenceIndex<TLink> : ISequenceIndex<TLink>
16      {
17          private static readonly EqualityComparer<TLink> _equalityComparer =
            ↪  EqualityComparer<TLink>.Default;
18          private readonly ISynchronizedLinks<TLink> _links;
19
20          /// <summary>
21          /// <para>
22          /// Initializes a new <see cref="SynchronizedSequenceIndex"/> instance.
23          /// </para>
24          /// <para></para>
25          /// </summary>
26          /// <param name="links">
27          /// <para>A links.</para>
28          /// <para></para>
29          /// </param>
30          [MethodImpl(MethodImplOptions.AggressiveInlining)]
31          public SynchronizedSequenceIndex(ISynchronizedLinks<TLink> links) => _links = links;
32
33          /// <summary>
34          /// <para>
35          /// Determines whether this instance add.
36          /// </para>
37          /// <para></para>
38          /// </summary>
39          /// <param name="sequence">
40          /// <para>The sequence.</para>
41          /// <para></para>
42          /// </param>
43          /// <returns>
44          /// <para>The indexed.</para>
45          /// <para></para>
46          /// </returns>
47          [MethodImpl(MethodImplOptions.AggressiveInlining)]
48          public bool Add(IList<TLink> sequence)
49          {
50              var indexed = true;
51              var i = sequence.Count;
52              var links = _links.Unsync;
53              _links.SyncRoot.ExecuteReadOperation(() =>
54              {
55                  while (--i >= 1 && (indexed =
                    ↪  !_equalityComparer.Equals(links.SearchOrDefault(sequence[i - 1],
                    ↪  sequence[i]), default))) { }
```

```
            });
            if (!indexed)
            {
                _links.SyncRoot.ExecuteWriteOperation(() =>
                {
                    for (; i >= 1; i--)
                    {
                        links.GetOrCreate(sequence[i - 1], sequence[i]);
                    }
                });
            }
            return indexed;
        }

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

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

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

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

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

```

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

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

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

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

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

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

        /// <summary>
        /// <para>
        /// Converts the decimal.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="@decimal">
```

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

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

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

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

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

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="RationalToDecimalConverter"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="links">
        /// <para>A links.</para>
        /// <para></para>
        /// </param>
        /// <param name="rawNumberSequenceToBigIntegerConverter">
        /// <para>A raw number sequence to big integer converter.</para>
        /// <para></para>
        /// </param>
```

```
42          public RationalToDecimalConverter(ILinks<TLink> links,
            ↪  RawNumberSequenceToBigIntegerConverter<TLink>
            ↪  rawNumberSequenceToBigIntegerConverter) : base(links)
43          {
44              RawNumberSequenceToBigIntegerConverter = rawNumberSequenceToBigIntegerConverter;
45          }
46
47          /// <summary>
48          /// <para>
49          /// Converts the rational number.
50          /// </para>
51          /// <para></para>
52          /// </summary>
53          /// <param name="rationalNumber">
54          /// <para>The rational number.</para>
55          /// <para></para>
56          /// </param>
57          /// <returns>
58          /// <para>The decimal</para>
59          /// <para></para>
60          /// </returns>
61          public decimal Convert(TLink rationalNumber)
62          {
63              var numerator = (decimal)RawNumberSequenceToBigIntegerConverter.Convert(_links.GetSo⌋
                ↪  urce(rationalNumber));
64              var denominator = (decimal)RawNumberSequenceToBigIntegerConverter.Convert(_links.Get⌋
                ↪  Target(rationalNumber));
65              return numerator / denominator;
66          }
67      }
68  }
```

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

```
1   using System.Collections.Generic;
2   using System.Numerics;
3   using 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{TLink}"/>
21      /// <seealso cref="IConverter{BigInteger, TLink}"/>
22      public class BigIntegerToRawNumberSequenceConverter<TLink> : LinksDecoratorBase<TLink>,
        ↪  IConverter<BigInteger, TLink>
23          where TLink : struct
24      {
25          /// <summary>
26          /// <para>
27          /// The max value.
28          /// </para>
29          /// <para></para>
30          /// </summary>
31          public static readonly TLink MaximumValue = NumericType<TLink>.MaxValue;
32          /// <summary>
33          /// <para>
34          /// The maximum value.
35          /// </para>
36          /// <para></para>
37          /// </summary>
38          public static readonly TLink 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<TLink> AddressToNumberConverter;
```

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

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

```
            var number = GetRawNumberParts(sign == -1 ? BigInteger.Negate(bigInteger) :
            ↪ bigInteger);
            var numberSequence = ListToSequenceConverter.Convert(number);
            return sign == -1 ? _links.GetOrCreate(NegativeNumberMarker, numberSequence) :
            ↪ numberSequence;
        }
    }
}
```

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

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

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

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

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

        /// <summary>
        /// <para>
        /// Converts the source.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="source">
        /// <para>The source.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The target</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TTarget Convert(TSource source)
        {
            var constants = Links.Constants;
            var externalReferencesRange = constants.ExternalReferencesRange;
            if (externalReferencesRange.HasValue &&
            ↪ externalReferencesRange.Value.Contains(source))
            {
```

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

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

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

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

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

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

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

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

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

```
using System;
using System.Collections.Generic;
using System.Numerics;
using Platform.Collections.Stacks;
using Platform.Converters;
using Platform.Data.Doublets.Decorators;
using Platform.Data.Doublets.Sequences.Walkers;
using Platform.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{TLink}"/>
    /// <seealso cref="IConverter{TLink, BigInteger}"/>
    public class RawNumberSequenceToBigIntegerConverter<TLink> : LinksDecoratorBase<TLink>,
    ↪  IConverter<TLink, BigInteger>
        where TLink : struct
    {
        /// <summary>
        /// <para>
        /// The default.
        /// </para>
        /// <para></para>
        /// </summary>
        public readonly EqualityComparer<TLink> EqualityComparer =
        ↪  EqualityComparer<TLink>.Default;
        /// <summary>
        /// <para>
        /// The number to address converter.
        /// </para>
        /// <para></para>
        /// </summary>
        public readonly IConverter<TLink, TLink> NumberToAddressConverter;
        /// <summary>
        /// <para>
        /// The left sequence walker.
        /// </para>
        /// <para></para>
        /// </summary>
        public readonly LeftSequenceWalker<TLink> LeftSequenceWalker;
        /// <summary>
        /// <para>
```

```
49          /// The negative number marker.
50          /// </para>
51          /// <para></para>
52          /// </summary>
53          public readonly TLink NegativeNumberMarker;
54
55          /// <summary>
56          /// <para>
57          /// Initializes a new <see cref="RawNumberSequenceToBigIntegerConverter"/> 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="numberToAddressConverter">
66          /// <para>A number to address converter.</para>
67          /// <para></para>
68          /// </param>
69          /// <param name="negativeNumberMarker">
70          /// <para>A negative number marker.</para>
71          /// <para></para>
72          /// </param>
73          public RawNumberSequenceToBigIntegerConverter(ILinks<TLink> links, IConverter<TLink,
            ↪ TLink> numberToAddressConverter, TLink negativeNumberMarker) : base(links)
74          {
75              NumberToAddressConverter = numberToAddressConverter;
76              LeftSequenceWalker = new(links, new DefaultStack<TLink>());
77              NegativeNumberMarker = negativeNumberMarker;
78          }
79
80          /// <summary>
81          /// <para>
82          /// Converts the big integer.
83          /// </para>
84          /// <para></para>
85          /// </summary>
86          /// <param name="bigInteger">
87          /// <para>The big integer.</para>
88          /// <para></para>
89          /// </param>
90          /// <exception cref="Exception">
91          /// <para>Raw number sequence cannot be empty.</para>
92          /// <para></para>
93          /// </exception>
94          /// <returns>
95          /// <para>The big integer</para>
96          /// <para></para>
97          /// </returns>
98          public BigInteger Convert(TLink bigInteger)
99          {
100             var sign = 1;
101             var bigIntegerSequence = bigInteger;
102             if (EqualityComparer.Equals(_links.GetSource(bigIntegerSequence),
                ↪ NegativeNumberMarker))
103             {
104                 sign = -1;
105                 bigIntegerSequence = _links.GetTarget(bigInteger);
106             }
107             using var enumerator = LeftSequenceWalker.Walk(bigIntegerSequence).GetEnumerator();
108             if (!enumerator.MoveNext())
109             {
110                 throw new Exception("Raw number sequence cannot be empty.");
111             }
112             var nextPart = NumberToAddressConverter.Convert(enumerator.Current);
113             BigInteger currentBigInt = new(nextPart.ToBytes());
114             while (enumerator.MoveNext())
115             {
116                 currentBigInt <<= NumericType<TLink>.BitsSize - 1;
117                 nextPart = NumberToAddressConverter.Convert(enumerator.Current);
118                 currentBigInt |= new BigInteger(nextPart.ToBytes());
119             }
120             return sign == -1 ? BigInteger.Negate(currentBigInt) : currentBigInt;
121         }
122     }
123 }
```

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

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

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

namespace Platform.Data.Doublets.Numbers.Unary
{
    /// <summary>
    /// <para>
    /// Represents the address to unary number converter.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="LinksOperatorBase{TLink}"/>
    /// <seealso cref="IConverter{TLink}"/>
    public class AddressToUnaryNumberConverter<TLink> : LinksOperatorBase<TLink>,
    ↪  IConverter<TLink>
    {
        private static readonly EqualityComparer<TLink> _equalityComparer =
        ↪  EqualityComparer<TLink>.Default;
        private static readonly TLink _zero = default;
        private static readonly TLink _one = Arithmetic.Increment(_zero);
        private readonly IConverter<int, TLink> _powerOf2ToUnaryNumberConverter;

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

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

```
        }
    }
}
```

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

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

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

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

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

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

```
        public TLink Convert(Doublet<TLink> doublet)
        {
            var links = _links;
            var link = links.SearchOrDefault(doublet.Source, doublet.Target);
            if (_equalityComparer.Equals(link, default))
            {
                throw new ArgumentException($"Link ({doublet}) not found.", nameof(doublet));
            }
            var frequency = _frequencyPropertyOperator.Get(link);
            if (_equalityComparer.Equals(frequency, default))
            {
                return default;
            }
            var frequencyNumber = links.GetSource(frequency);
            return _unaryNumberToAddressConverter.Convert(frequencyNumber);
        }
    }
}
```

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

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

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

namespace Platform.Data.Doublets.Numbers.Unary
{
    /// <summary>
    /// <para>
    /// Represents the power of to unary number converter.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="LinksOperatorBase{TLink}"/>
    /// <seealso cref="IConverter{int, TLink}"/>
    public class PowerOf2ToUnaryNumberConverter<TLink> : LinksOperatorBase<TLink>,
    ↪  IConverter<int, TLink>
    {
        private static readonly EqualityComparer<TLink> _equalityComparer =
        ↪  EqualityComparer<TLink>.Default;
        private readonly TLink[] _unaryNumberPowersOf2;

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

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

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

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

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

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

namespace Platform.Data.Doublets.Numbers.Unary
{
    /// <summary>
    /// <para>
    /// Represents the unary number to address add operation converter.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="LinksOperatorBase{TLink}"/>
    /// <seealso cref="IConverter{TLink}"/>
    public class UnaryNumberToAddressAddOperationConverter<TLink> : LinksOperatorBase<TLink>, IConverter<TLink>
    {
        private static readonly EqualityComparer<TLink> _equalityComparer = EqualityComparer<TLink>.Default;
        private static readonly UncheckedConverter<TLink, ulong> _addressToUInt64Converter = UncheckedConverter<TLink, ulong>.Default;
        private static readonly UncheckedConverter<ulong, TLink> _uInt64ToAddressConverter = UncheckedConverter<ulong, TLink>.Default;
        private static readonly TLink _zero = default;
        private static readonly TLink _one = Arithmetic.Increment(_zero);
        private readonly Dictionary<TLink, TLink> _unaryToUInt64;
        private readonly TLink _unaryOne;

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

        /// <summary>
        /// <para>
        /// Converts the unary number.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="unaryNumber">
```

```
        /// <para>The unary number.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TLink Convert(TLink unaryNumber)
        {
            if (_equalityComparer.Equals(unaryNumber, default))
            {
                return default;
            }
            if (_equalityComparer.Equals(unaryNumber, _unaryOne))
            {
                return _one;
            }
            var links = _links;
            var source = links.GetSource(unaryNumber);
            var target = links.GetTarget(unaryNumber);
            if (_equalityComparer.Equals(source, target))
            {
                return _unaryToUInt64[unaryNumber];
            }
            else
            {
                var result = _unaryToUInt64[source];
                TLink lastValue;
                while (!_unaryToUInt64.TryGetValue(target, out lastValue))
                {
                    source = links.GetSource(target);
                    result = Arithmetic<TLink>.Add(result, _unaryToUInt64[source]);
                    target = links.GetTarget(target);
                }
                result = Arithmetic<TLink>.Add(result, lastValue);
                return result;
            }
        }
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static Dictionary<TLink, TLink> CreateUnaryToUInt64Dictionary(ILinks<TLink>
        ↪  links, TLink unaryOne)
        {
            var unaryToUInt64 = new Dictionary<TLink, TLink>
            {
                { unaryOne, _one }
            };
            var unary = unaryOne;
            var number = _one;
            for (var i = 1; i < 64; i++)
            {
                unary = links.GetOrCreate(unary, unary);
                number = Double(number);
                unaryToUInt64.Add(unary, number);
            }
            return unaryToUInt64;
        }
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static TLink Double(TLink number) =>
        ↪  _uInt64ToAddressConverter.Convert(_addressToUInt64Converter.Convert(number) * 2UL);
    }
}
```

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

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

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

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

```
16      /// </summary>
17      /// <seealso cref="LinksOperatorBase{TLink}"/>
18      /// <seealso cref="IConverter{TLink}"/>
19      public class UnaryNumberToAddressOrOperationConverter<TLink> : LinksOperatorBase<TLink>,
        ↪  IConverter<TLink>
20      {
21          private static readonly EqualityComparer<TLink> _equalityComparer =
            ↪  EqualityComparer<TLink>.Default;
22          private static readonly TLink _zero = default;
23          private static readonly TLink _one = Arithmetic.Increment(_zero);
24          private readonly IDictionary<TLink, int> _unaryNumberPowerOf2Indicies;
25
26          /// <summary>
27          /// <para>
28          /// Initializes a new <see cref="UnaryNumberToAddressOrOperationConverter"/> 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 UnaryNumberToAddressOrOperationConverter(ILinks<TLink> links, IConverter<int,
            ↪  TLink> powerOf2ToUnaryNumberConverter) : base(links) => _unaryNumberPowerOf2Indicies
            ↪  = CreateUnaryNumberPowerOf2IndiciesDictionary(powerOf2ToUnaryNumberConverter);
42
43          /// <summary>
44          /// <para>
45          /// Converts the source number.
46          /// </para>
47          /// <para></para>
48          /// </summary>
49          /// <param name="sourceNumber">
50          /// <para>The source 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 TLink Convert(TLink sourceNumber)
59          {
60              var links = _links;
61              var nullConstant = links.Constants.Null;
62              var source = sourceNumber;
63              var target = nullConstant;
64              if (!_equalityComparer.Equals(source, nullConstant))
65              {
66                  while (true)
67                  {
68                      if (_unaryNumberPowerOf2Indicies.TryGetValue(source, out int powerOf2Index))
69                      {
70                          SetBit(ref target, powerOf2Index);
71                          break;
72                      }
73                      else
74                      {
75                          powerOf2Index = _unaryNumberPowerOf2Indicies[links.GetSource(source)];
76                          SetBit(ref target, powerOf2Index);
77                          source = links.GetTarget(source);
78                      }
79                  }
80              }
81              return target;
82          }
83          [MethodImpl(MethodImplOptions.AggressiveInlining)]
84          private static Dictionary<TLink, int>
            ↪  CreateUnaryNumberPowerOf2IndiciesDictionary(IConverter<int, TLink>
            ↪  powerOf2ToUnaryNumberConverter)
85          {
86              var unaryNumberPowerOf2Indicies = new Dictionary<TLink, int>();
87              for (int i = 0; i < NumericType<TLink>.BitsSize; i++)
```

```
            {
                unaryNumberPowerOf2Indicies.Add(powerOf2ToUnaryNumberConverter.Convert(i), i);
            }
            return unaryNumberPowerOf2Indicies;
        }
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static void SetBit(ref TLink target, int powerOf2Index) => target =
        ↪ Bit.Or(target, Bit.ShiftLeft(_one, powerOf2Index));
    }
}
```

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

```
using System;
using System.Collections.Generic;
using System.Runtime.CompilerServices;
using System.Linq;
using System.Text;
using Platform.Collections;
using Platform.Collections.Sets;
using Platform.Collections.Stacks;
using Platform.Data.Exceptions;
using Platform.Data.Sequences;
using Platform.Data.Doublets.Sequences.Frequencies.Counters;
using Platform.Data.Doublets.Sequences.Walkers;
using LinkIndex = System.UInt64;
using Stack = System.Collections.Generic.Stack<ulong>;

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

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

        /// <remarks>
        /// Number of links that is needed to generate all variants for
        /// sequence of length N corresponds to https://oeis.org/A014143/list sequence.
        /// </remarks>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public ulong[] CreateAllVariants2(ulong[] sequence)
        {
            return _sync.ExecuteWriteOperation(() =>
            {
                if (sequence.IsNullOrEmpty())
                {
                    return Array.Empty<ulong>();
                }
                Links.EnsureLinkExists(sequence);
                if (sequence.Length == 1)
                {
                    return sequence;
                }
                return CreateAllVariants2Core(sequence, 0, (ulong)sequence.Length - 1);
            });
        }
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private ulong[] CreateAllVariants2Core(ulong[] sequence, ulong startAt, ulong stopAt)
        {
            if ((stopAt - startAt) == 0)
            {
                return new[] { sequence[startAt] };
            }
            if ((stopAt - startAt) == 1)
            {
                return new[] { Links.Unsync.GetOrCreate(sequence[startAt], sequence[stopAt]) };
            }
            var variants = new ulong[Platform.Numbers.Math.Catalan(stopAt - startAt)];
            var last = 0;
            for (var splitter = startAt; splitter < stopAt; splitter++)
            {
                var left = CreateAllVariants2Core(sequence, startAt, splitter);
                var right = CreateAllVariants2Core(sequence, splitter + 1, stopAt);
```

```
                for (var i = 0; i < left.Length; i++)
                {
                    for (var j = 0; j < right.Length; j++)
                    {
                        var variant = Links.Unsync.GetOrCreate(left[i], right[j]);
                        if (variant == Constants.Null)
                        {
                            throw new NotImplementedException("Creation cancellation is not
                            ↪  implemented.");
                        }
                        variants[last++] = variant;
                    }
                }
            }
            return variants;
        }

        /// <summary>
        /// <para>
        /// Creates the all variants 1 using the specified sequence.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="sequence">
        /// <para>The sequence.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>A list of ulong</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public List<ulong> CreateAllVariants1(params ulong[] sequence)
        {
            return _sync.ExecuteWriteOperation(() =>
            {
                if (sequence.IsNullOrEmpty())
                {
                    return new List<ulong>();
                }
                Links.Unsync.EnsureLinkExists(sequence);
                if (sequence.Length == 1)
                {
                    return new List<ulong> { sequence[0] };
                }
                var results = new
                ↪  List<ulong>((int)Platform.Numbers.Math.Catalan((ulong)sequence.Length));
                return CreateAllVariants1Core(sequence, results);
            });
        }
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private List<ulong> CreateAllVariants1Core(ulong[] sequence, List<ulong> results)
        {
            if (sequence.Length == 2)
            {
                var link = Links.Unsync.GetOrCreate(sequence[0], sequence[1]);
                if (link == Constants.Null)
                {
                    throw new NotImplementedException("Creation cancellation is not
                    ↪  implemented.");
                }
                results.Add(link);
                return results;
            }
            var innerSequenceLength = sequence.Length - 1;
            var innerSequence = new ulong[innerSequenceLength];
            for (var li = 0; li < innerSequenceLength; li++)
            {
                var link = Links.Unsync.GetOrCreate(sequence[li], sequence[li + 1]);
                if (link == Constants.Null)
                {
                    throw new NotImplementedException("Creation cancellation is not
                    ↪  implemented.");
                }
                for (var isi = 0; isi < li; isi++)
                {
                    innerSequence[isi] = sequence[isi];
                }
```

```
                innerSequence[li] = link;
                for (var isi = li + 1; isi < innerSequenceLength; isi++)
                {
                    innerSequence[isi] = sequence[isi + 1];
                }
                CreateAllVariants1Core(innerSequence, results);
            }
            return results;
        }

        #endregion

        /// <summary>
        /// <para>
        /// Eaches the 1 using the specified sequence.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="sequence">
        /// <para>The sequence.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The visited links.</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public HashSet<ulong> Each1(params ulong[] sequence)
        {
            var visitedLinks = new HashSet<ulong>(); // Заменить на bitstring
            Each1(link =>
            {
                if (!visitedLinks.Contains(link))
                {
                    visitedLinks.Add(link); // изучить почему случаются повторы
                }
                return true;
            }, sequence);
            return visitedLinks;
        }
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private void Each1(Func<ulong, bool> handler, params ulong[] sequence)
        {
            if (sequence.Length == 2)
            {
                Links.Unsync.Each(sequence[0], sequence[1], handler);
            }
            else
            {
                var innerSequenceLength = sequence.Length - 1;
                for (var li = 0; li < innerSequenceLength; li++)
                {
                    var left = sequence[li];
                    var right = sequence[li + 1];
                    if (left == 0 && right == 0)
                    {
                        continue;
                    }
                    var linkIndex = li;
                    ulong[] innerSequence = null;
                    Links.Unsync.Each(doublet =>
                    {
                        if (innerSequence == null)
                        {
                            innerSequence = new ulong[innerSequenceLength];
                            for (var isi = 0; isi < linkIndex; isi++)
                            {
                                innerSequence[isi] = sequence[isi];
                            }
                            for (var isi = linkIndex + 1; isi < innerSequenceLength; isi++)
                            {
                                innerSequence[isi] = sequence[isi + 1];
                            }
                        }
                        innerSequence[linkIndex] = doublet[Constants.IndexPart];
                        Each1(handler, innerSequence);
                        return Constants.Continue;
                    }, Constants.Any, left, right);
```

```
                }
            }
        }

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

        /// <summary>
        /// <para>
        /// Eaches the part using the specified handler.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="handler">
        /// <para>The handler.</para>
        /// <para></para>
        /// </param>
        /// <param name="sequence">
        /// <para>The sequence.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void EachPart(Func<IList<LinkIndex>, LinkIndex> handler, params ulong[] sequence)
        {
            var visitedLinks = new HashSet<ulong>(); // Заменить на bitstring
            EachPartCore(link =>
            {
                var linkIndex = link[Constants.IndexPart];
                if (!visitedLinks.Contains(linkIndex))
                {
                    visitedLinks.Add(linkIndex); // изучить почему случаются повторы
                    return handler(new LinkAddress<LinkIndex>(linkIndex));
                }
                return Constants.Continue;
            }, sequence);
        }
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private void EachPartCore(Func<IList<LinkIndex>, LinkIndex> handler, params ulong[]
        ↪  sequence)
        {
            if (sequence.IsNullOrEmpty())
            {
                return;
            }
            Links.EnsureLinkIsAnyOrExists(sequence);
            if (sequence.Length == 1)
            {
                var link = sequence[0];
                if (link > 0)
                {
                    handler(new LinkAddress<LinkIndex>(link));
```

```
                }
                else
                {
                    Links.Each(Constants.Any, Constants.Any, handler);
                }
            }
            else if (sequence.Length == 2)
            {
                //_links.Each(sequence[0], sequence[1], handler);
                //  o_|      x_o ...
                // x_|        |___|
                Links.Each(sequence[1], Constants.Any, doublet =>
                {
                    var match = Links.SearchOrDefault(sequence[0], doublet);
                    if (match != Constants.Null)
                    {
                        handler(new LinkAddress<LinkIndex>(match));
                    }
                    return true;
                });
                // |_x      ... x_o
                //  |_o      |___|
                Links.Each(Constants.Any, sequence[0], doublet =>
                {
                    var match = Links.SearchOrDefault(doublet, sequence[1]);
                    if (match != 0)
                    {
                        handler(new LinkAddress<LinkIndex>(match));
                    }
                    return true;
                });
                //           ._x o_.
                //            |___|
                PartialStepRight(x => handler(x), sequence[0], sequence[1]);
            }
            else
            {
                throw new NotImplementedException();
            }
        }
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private void PartialStepRight(Action<IList<LinkIndex>> handler, ulong left, ulong right)
        {
            Links.Unsync.Each(Constants.Any, left, doublet =>
            {
                StepRight(handler, doublet, right);
                if (left != doublet)
                {
                    PartialStepRight(handler, doublet, right);
                }
                return true;
            });
        }
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private void StepRight(Action<IList<LinkIndex>> handler, ulong left, ulong right)
        {
            Links.Unsync.Each(left, Constants.Any, rightStep =>
            {
                TryStepRightUp(handler, right, rightStep);
                return true;
            });
        }
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private void TryStepRightUp(Action<IList<LinkIndex>> handler, ulong right, ulong
        ↪  stepFrom)
        {
            var upStep = stepFrom;
            var firstSource = Links.Unsync.GetTarget(upStep);
            while (firstSource != right && firstSource != upStep)
            {
                upStep = firstSource;
                firstSource = Links.Unsync.GetSource(upStep);
            }
            if (firstSource == right)
            {
                handler(new LinkAddress<LinkIndex>(stepFrom));
            }
        }
```

```
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.ExecuteReadOperation(() =>
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.ExecuteReadOperation(() =>
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.ExecuteReadOperation(() => 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.ExecuteReadOperation(() => 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                         }
```

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

        /// <summary>
        /// <para>
        /// Gets the all partially matching sequences 0 using the specified sequence.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="sequence">
        /// <para>The sequence.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>A list of ulong</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public List<ulong> GetAllPartiallyMatchingSequences0(params ulong[] sequence)
        {
            return _sync.ExecuteReadOperation(() =>
            {
                if (sequence.Length > 0)
                {
                    Links.EnsureLinkExists(sequence);
                    var results = new HashSet<ulong>();
                    for (var i = 0; i < sequence.Length; i++)
                    {
                        AllUsagesCore(sequence[i], results);
                    }
                    var filteredResults = new List<ulong>();
                    var linksInSequence = new HashSet<ulong>(sequence);
                    foreach (var result in results)
                    {
                        var filterPosition = -1;
                        StopableSequenceWalker.WalkRight(result, Links.Unsync.GetSource,
                        ↪  Links.Unsync.GetTarget,
                            x => linksInSequence.Contains(x) || Links.Unsync.GetTarget(x) == x,
                            ↪  x =>
                            {
                                if (filterPosition == (sequence.Length - 1))
                                {
                                    return false;
                                }
                                if (filterPosition >= 0)
                                {
                                    if (x == sequence[filterPosition + 1])
                                    {
                                        filterPosition++;
                                    }
                                    else
                                    {
                                        return false;
                                    }
                                }
                                if (filterPosition < 0)
                                {
                                    if (x == sequence[0])
                                    {
                                        filterPosition = 0;
                                    }
                                }
                                return true;
                            });
                        if (filterPosition == (sequence.Length - 1))
```

```
818                     {
819                         filteredResults.Add(result);
820                     }
821                 }
822                 return filteredResults;
823             }
824             return new List<ulong>();
825         });
826     }
827 
828     /// <summary>
829     /// <para>
830     /// Gets the all partially matching sequences 1 using the specified sequence.
831     /// </para>
832     /// <para></para>
833     /// </summary>
834     /// <param name="sequence">
835     /// <para>The sequence.</para>
836     /// <para></para>
837     /// </param>
838     /// <returns>
839     /// <para>A hash set of ulong</para>
840     /// <para></para>
841     /// </returns>
842     [MethodImpl(MethodImplOptions.AggressiveInlining)]
843     public HashSet<ulong> GetAllPartiallyMatchingSequences1(params ulong[] sequence)
844     {
845         return _sync.ExecuteReadOperation(() =>
846         {
847             if (sequence.Length > 0)
848             {
849                 Links.EnsureLinkExists(sequence);
850                 var results = new HashSet<ulong>();
851                 for (var i = 0; i < sequence.Length; i++)
852                 {
853                     AllUsagesCore(sequence[i], results);
854                 }
855                 var filteredResults = new HashSet<ulong>();
856                 var matcher = new Matcher(this, sequence, filteredResults, null);
857                 matcher.AddAllPartialMatchedToResults(results);
858                 return filteredResults;
859             }
860             return new HashSet<ulong>();
861         });
862     }
863 
864     /// <summary>
865     /// <para>
866     /// Determines whether this instance get all partially matching sequences 2.
867     /// </para>
868     /// <para></para>
869     /// </summary>
870     /// <param name="handler">
871     /// <para>The handler.</para>
872     /// <para></para>
873     /// </param>
874     /// <param name="sequence">
875     /// <para>The sequence.</para>
876     /// <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.ExecuteReadOperation(() =>
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++)
```

```
                {
                    if (!AllUsagesCore1(sequence[i], results, matcher.HandlePartialMatched))
                    {
                        return false;
                    }
                }
                return true;
            }
            return true;
        });
    }

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

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

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

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

    //            firstResults.IntersectWith(lastResults);

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

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

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

    /// <summary>
    /// <para>
    /// Gets the all partially matching sequences 3 using the specified sequence.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <param name="sequence">
    /// <para>The sequence.</para>
    /// <para></para>
    /// </param>
    /// <returns>
    /// <para>A hash set of ulong</para>
    /// <para></para>
    /// </returns>
    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    public HashSet<ulong> GetAllPartiallyMatchingSequences3(params ulong[] sequence)
    {
        return _sync.ExecuteReadOperation(() =>
        {
            if (sequence.Length > 0)
            {
                ILinksExtensions.EnsureLinkIsAnyOrExists(Links, sequence);
                var firstResults = new HashSet<ulong>();
                var lastResults = new HashSet<ulong>();
                var first = sequence.First(x => x != Constants.Any);
                var last = sequence.Last(x => x != Constants.Any);
                AllUsagesCore(first, firstResults);
                AllUsagesCore(last, lastResults);
                firstResults.IntersectWith(lastResults);
                //for (var i = 0; i < sequence.Length; i++)
                //    AllUsagesCore(sequence[i], results);
                var filteredResults = new HashSet<ulong>();
                var matcher = new Matcher(this, sequence, filteredResults, null);
                matcher.AddAllPartialMatchedToResults(firstResults);
                return filteredResults;
```

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

    /// <summary>
    /// <para>
    /// Gets the all partially matching sequences 4 using the specified read as elements.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <param name="readAsElements">
    /// <para>The read as elements.</para>
    /// <para></para>
    /// </param>
    /// <param name="sequence">
    /// <para>The sequence.</para>
    /// <para></para>
    /// </param>
    /// <returns>
    /// <para>A hash set of ulong</para>
    /// <para></para>
    /// </returns>
    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    public HashSet<ulong> GetAllPartiallyMatchingSequences4(HashSet<ulong> readAsElements,
    ↪  IList<ulong> sequence)
    {
        return _sync.ExecuteReadOperation(() =>
        {
            if (sequence.Count > 0)
            {
                Links.EnsureLinkExists(sequence);
                var results = new HashSet<LinkIndex>();
                //var nextResults = new HashSet<ulong>();
                //for (var i = 0; i < sequence.Length; i++)
                //{
                //    AllUsagesCore(sequence[i], nextResults);
                //    if (results.IsNullOrEmpty())
                //    {
                //        results = nextResults;
                //        nextResults = new HashSet<ulong>();
                //    }
                //    else
                //    {
                //        results.IntersectWith(nextResults);
                //        nextResults.Clear();
                //    }
                //}
                var collector1 = new AllUsagesCollector1(Links.Unsync, results);
                collector1.Collect(Links.Unsync.GetLink(sequence[0]));
                var next = new HashSet<ulong>();
                for (var i = 1; i < sequence.Count; i++)
                {
                    var collector = new AllUsagesCollector1(Links.Unsync, next);
                    collector.Collect(Links.Unsync.GetLink(sequence[i]));

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

    // Does not work
    //public HashSet<ulong> GetAllPartiallyMatchingSequences5(HashSet<ulong> readAsElements,
    ↪  params ulong[] sequence)
    //{
    //    var visited = new HashSet<ulong>();
    //    var results = new HashSet<ulong>();
    //    var matcher = new Matcher(this, sequence, visited, x => { results.Add(x); return
    ↪  true; }, readAsElements);
```

```
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.ExecuteReadOperation(() =>
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)
```

```
                    //////        matches.Add(results);
                    //////    else
                    //////        return results;
                    //////    if (matches.Count == 2)
                    //////    {
                    //////        var merged = new List<ulong>();
                    //////        for (var j = 0; j < matches[0].Count; j++)
                    //////            for (var k = 0; k < matches[1].Count; k++)
                    //////                CloseInnerConnections(merged.Add, matches[0][j],
                    ↪  matches[1][k]);
                    //////        if (merged.Count > 0)
                    //////            matches = new List<List<ulong>> { merged };
                    //////        else
                    //////            return new List<ulong>();
                    //////    }
                    //////}
                    //////if (matches.Count > 0)
                    //////{
                    //////    var usages = new HashSet<ulong>();
                    //////    for (int i = 0; i < sequence.Length; i++)
                    //////    {
                    //////        AllUsagesCore(sequence[i], usages);
                    //////    }
                    //////    //for (int i = 0; i < matches[0].Count; i++)
                    //////    //    AllUsagesCore(matches[0][i], usages);
                    //////    //usages.UnionWith(matches[0]);
                    //////    return usages.ToList();
                    //////}
                    var firstLinkUsages = new HashSet<ulong>();
                    AllUsagesCore(sequence[0], firstLinkUsages);
                    firstLinkUsages.Add(sequence[0]);
                    //var previousMatchings = firstLinkUsages.ToList(); //new List<ulong>() {
                    ↪  sequence[0] }; // or all sequences, containing this element?
                    //return GetAllPartiallyMatchingSequencesCore(sequence, firstLinkUsages,
                    ↪  1).ToList();
                    var results = new HashSet<ulong>();
                    foreach (var match in GetAllPartiallyMatchingSequencesCore(sequence,
                    ↪  firstLinkUsages, 1))
                    {
                        AllUsagesCore(match, results);
                    }
                    return results.ToList();
                }
                return new List<ulong>();
            });
        }

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

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

```
        }

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

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

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

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

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

            /// <summary>
            /// <para>
            /// Initializes a new <see cref="AllUsagesCalculator"/> instance.
            /// </para>
            /// <para></para>
            /// </summary>
            /// <param name="links">
            /// <para>A links.</para>
            /// <para></para>
            /// </param>
            /// <param name="totals">
            /// <para>A totals.</para>
            /// <para></para>
            /// </param>
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
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;
```

```
            }
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            private bool CalculateCore(ulong link)
            {
                // TODO: Проработать защиту от зацикливания
                // Основано на SequenceWalker.WalkLeft
                Func<ulong, ulong> getSource = _links.Unsync.GetSource;
                Func<ulong, ulong> getTarget = _links.Unsync.GetTarget;
                Func<ulong, bool> isElement = IsElement;
                void visitLeaf(ulong parent)
                {
                    if (link != parent)
                    {
                        _totals[parent]++;
                    }
                }
                void visitNode(ulong parent)
                {
                    if (link != parent)
                    {
                        _totals[parent]++;
                    }
                }
                var stack = new Stack();
                var element = link;
                if (isElement(element))
                {
                    visitLeaf(element);
                }
                else
                {
                    while (true)
                    {
                        if (isElement(element))
                        {
                            if (stack.Count == 0)
                            {
                                break;
                            }
                            element = stack.Pop();
                            var source = getSource(element);
                            var target = getTarget(element);
                            // Обработка элемента
                            if (isElement(target))
                            {
                                visitLeaf(target);
                            }
                            if (isElement(source))
                            {
                                visitLeaf(source);
                            }
                            element = source;
                        }
                        else
                        {
                            stack.Push(element);
                            visitNode(element);
                            element = getTarget(element);
                        }
                    }
                }
                _totals[link]++;
                return true;
            }
        }
        private class AllUsagesCollector
        {
            private readonly ILinks<ulong> _links;
            private readonly HashSet<ulong> _usages;

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

```
            /// </param>
            /// <param name="usages">
            /// <para>A usages.</para>
            /// <para></para>
            /// </param>
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public AllUsagesCollector(ILinks<ulong> links, HashSet<ulong> usages)
            {
                _links = links;
                _usages = usages;
            }

            /// <summary>
            /// <para>
            /// Determines whether this instance collect.
            /// </para>
            /// <para></para>
            /// </summary>
            /// <param name="link">
            /// <para>The link.</para>
            /// <para></para>
            /// </param>
            /// <returns>
            /// <para>The bool</para>
            /// <para></para>
            /// </returns>
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public bool Collect(ulong link)
            {
                if (_usages.Add(link))
                {
                    _links.Each(link, _links.Constants.Any, Collect);
                    _links.Each(_links.Constants.Any, link, Collect);
                }
                return true;
            }
        }
        private class AllUsagesCollector1
        {
            private readonly ILinks<ulong> _links;
            private readonly HashSet<ulong> _usages;
            private readonly ulong _continue;

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

            /// <summary>
            /// <para>
            /// Collects the link.
            /// </para>
            /// <para></para>
            /// </summary>
            /// <param name="link">
            /// <para>The link.</para>
            /// <para></para>
            /// </param>
            /// <returns>
            /// <para>The continue.</para>
            /// <para></para>
            /// </returns>
```

```
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public ulong Collect(IList<ulong> link)
            {
                var linkIndex = _links.GetIndex(link);
                if (_usages.Add(linkIndex))
                {
                    _links.Each(Collect, _links.Constants.Any, linkIndex);
                }
                return _continue;
            }
        }
        private class AllUsagesCollector2
        {
            private readonly ILinks<ulong> _links;
            private readonly BitString _usages;

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

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

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

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

            /// <summary>
            /// <para>
            /// Determines whether this instance collect.
            /// </para>
            /// <para></para>
            /// </summary>
            /// <param name="link">
            /// <para>The link.</para>
            /// <para></para>
            /// </param>
            /// <returns>
            /// <para>The bool</para>
            /// <para></para>
            /// </returns>
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public bool Collect(ulong link)
            {
                if (_enter.Add(link))
                {
                    if (_intersectWith.Contains(link))
                    {
                        _usages.Add(link);
                    }
                    _links.Unsync.Each(link, _links.Constants.Any, Collect);
                    _links.Unsync.Each(_links.Constants.Any, link, Collect);
                }
                return true;
            }
        }
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private void CloseInnerConnections(Action<IList<LinkIndex>> handler, ulong left, ulong
        ↪  right)
        {
            TryStepLeftUp(handler, left, right);
            TryStepRightUp(handler, right, left);
        }
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private void AllCloseConnections(Action<IList<LinkIndex>> handler, ulong left, ulong
        ↪  right)
        {
            // Direct
            if (left == right)
            {
                handler(new LinkAddress<LinkIndex>(left));
            }
            var doublet = Links.Unsync.SearchOrDefault(left, right);
            if (doublet != Constants.Null)
            {
                handler(new LinkAddress<LinkIndex>(doublet));
            }
            // Inner
            CloseInnerConnections(handler, left, right);
            // Outer
            StepLeft(handler, left, right);
            StepRight(handler, left, right);
            PartialStepRight(handler, left, right);
            PartialStepLeft(handler, left, right);
        }
        [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.ExecuteReadOperation(() =>
1796                {
1797                    patternSequence = Simplify(patternSequence);
1798                    if (patternSequence.Length > 0)
```

```
            {
                EnsureEachLinkIsAnyOrZeroOrManyOrExists(Links, patternSequence);
                var uniqueSequenceElements = new HashSet<ulong>();
                for (var i = 0; i < patternSequence.Length; i++)
                {
                    if (patternSequence[i] != Constants.Any && patternSequence[i] !=
                    ↪  ZeroOrMany)
                    {
                        uniqueSequenceElements.Add(patternSequence[i]);
                    }
                }
                var results = new HashSet<ulong>();
                foreach (var uniqueSequenceElement in uniqueSequenceElements)
                {
                    AllUsagesCore(uniqueSequenceElement, results);
                }
                var filteredResults = new HashSet<ulong>();
                var matcher = new PatternMatcher(this, patternSequence, filteredResults);
                matcher.AddAllPatternMatchedToResults(results);
                return filteredResults;
            }
            return new HashSet<ulong>();
        });
    }

    // Найти все возможные связи между указанным списком связей.
    // Находит связи между всеми указанными связями в любом порядке.
    // TODO: решить что делать с повторами (когда одни и те же элементы встречаются
    ↪  несколько раз в последовательности)
    /// <summary>
    /// <para>
    /// Gets the all connections using the specified links to connect.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <param name="linksToConnect">
    /// <para>The links to connect.</para>
    /// <para></para>
    /// </param>
    /// <returns>
    /// <para>A hash set of ulong</para>
    /// <para></para>
    /// </returns>
    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    public HashSet<ulong> GetAllConnections(params ulong[] linksToConnect)
    {
        return _sync.ExecuteReadOperation(() =>
        {
            var results = new HashSet<ulong>();
            if (linksToConnect.Length > 0)
            {
                Links.EnsureLinkExists(linksToConnect);
                AllUsagesCore(linksToConnect[0], results);
                for (var i = 1; i < linksToConnect.Length; i++)
                {
                    var next = new HashSet<ulong>();
                    AllUsagesCore(linksToConnect[i], next);
                    results.IntersectWith(next);
                }
            }
            return results;
        });
    }

    /// <summary>
    /// <para>
    /// Gets the all connections 1 using the specified links to connect.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <param name="linksToConnect">
    /// <para>The links to connect.</para>
    /// <para></para>
    /// </param>
    /// <returns>
    /// <para>A hash set of ulong</para>
    /// <para></para>
    /// </returns>
```

```
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public HashSet<ulong> GetAllConnections1(params ulong[] linksToConnect)
        {
            return _sync.ExecuteReadOperation(() =>
            {
                var results = new HashSet<ulong>();
                if (linksToConnect.Length > 0)
                {
                    Links.EnsureLinkExists(linksToConnect);
                    var collector1 = new AllUsagesCollector(Links.Unsync, results);
                    collector1.Collect(linksToConnect[0]);
                    var next = new HashSet<ulong>();
                    for (var i = 1; i < linksToConnect.Length; i++)
                    {
                        var collector = new AllUsagesCollector(Links.Unsync, next);
                        collector.Collect(linksToConnect[i]);
                        results.IntersectWith(next);
                        next.Clear();
                    }
                }
                return results;
            });
        }

        /// <summary>
        /// <para>
        /// Gets the all connections 2 using the specified links to connect.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="linksToConnect">
        /// <para>The links to connect.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>A hash set of ulong</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public HashSet<ulong> GetAllConnections2(params ulong[] linksToConnect)
        {
            return _sync.ExecuteReadOperation(() =>
            {
                var results = new HashSet<ulong>();
                if (linksToConnect.Length > 0)
                {
                    Links.EnsureLinkExists(linksToConnect);
                    var collector1 = new AllUsagesCollector(Links, results);
                    collector1.Collect(linksToConnect[0]);
                    //AllUsagesCore(linksToConnect[0], results);
                    for (var i = 1; i < linksToConnect.Length; i++)
                    {
                        var next = new HashSet<ulong>();
                        var collector = new AllUsagesIntersectingCollector(Links, results, next);
                        collector.Collect(linksToConnect[i]);
                        //AllUsagesCore(linksToConnect[i], next);
                        //results.IntersectWith(next);
                        results = next;
                    }
                }
                return results;
            });
        }

        /// <summary>
        /// <para>
        /// Gets the all connections 3 using the specified links to connect.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="linksToConnect">
        /// <para>The links to connect.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>A list of ulong</para>
        /// <para></para>
        /// </returns>
```

```
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public List<ulong> GetAllConnections3(params ulong[] linksToConnect)
        {
            return _sync.ExecuteReadOperation(() =>
            {
                var results = new BitString((long)Links.Unsync.Count() + 1); // new
                ↪  BitArray((int)_links.Total + 1);
                if (linksToConnect.Length > 0)
                {
                    Links.EnsureLinkExists(linksToConnect);
                    var collector1 = new AllUsagesCollector2(Links.Unsync, results);
                    collector1.Collect(linksToConnect[0]);
                    for (var i = 1; i < linksToConnect.Length; i++)
                    {
                        var next = new BitString((long)Links.Unsync.Count() + 1); //new
                        ↪  BitArray((int)_links.Total + 1);
                        var collector = new AllUsagesCollector2(Links.Unsync, next);
                        collector.Collect(linksToConnect[i]);
                        results = results.And(next);
                    }
                }
                return results.GetSetUInt64Indices();
            });
        }
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static ulong[] Simplify(ulong[] sequence)
        {
            // Считаем новый размер последовательности
            long newLength = 0;
            var zeroOrManyStepped = false;
            for (var i = 0; i < sequence.Length; i++)
            {
                if (sequence[i] == ZeroOrMany)
                {
                    if (zeroOrManyStepped)
                    {
                        continue;
                    }
                    zeroOrManyStepped = true;
                }
                else
                {
                    //if (zeroOrManyStepped) Is it efficient?
                    zeroOrManyStepped = false;
                }
                newLength++;
            }
            // Строим новую последовательность
            zeroOrManyStepped = false;
            var newSequence = new ulong[newLength];
            long j = 0;
            for (var i = 0; i < sequence.Length; i++)
            {
                //var current = zeroOrManyStepped;
                //zeroOrManyStepped = patternSequence[i] == zeroOrMany;
                //if (current && zeroOrManyStepped)
                //    continue;
                //var newZeroOrManyStepped = patternSequence[i] == zeroOrMany;
                //if (zeroOrManyStepped && newZeroOrManyStepped)
                //    continue;
                //zeroOrManyStepped = newZeroOrManyStepped;
                if (sequence[i] == ZeroOrMany)
                {
                    if (zeroOrManyStepped)
                    {
                        continue;
                    }
                    zeroOrManyStepped = true;
                }
                else
                {
                    //if (zeroOrManyStepped) Is it efficient?
                    zeroOrManyStepped = false;
                }
                newSequence[j++] = sequence[i];
            }
            return newSequence;
        }
```

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

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

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

        #region From Triplets

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

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

```
2106            return results;
2107        }
2108        [MethodImpl(MethodImplOptions.AggressiveInlining)]
2109        private void CollectMatchingSequences(ulong leftLink, int leftBound, ulong[]
            ↪ middleLinks, ulong rightLink, int rightBound, ref List<ulong> results)
2110        {
2111            var leftLinkTotalReferers = Links.Unsync.Count(leftLink);
2112            var rightLinkTotalReferers = Links.Unsync.Count(rightLink);
2113            if (leftLinkTotalReferers <= rightLinkTotalReferers)
2114            {
2115                var nextLeftLink = middleLinks[leftBound];
2116                var elements = GetRightElements(leftLink, nextLeftLink);
2117                if (leftBound <= rightBound)
2118                {
2119                    for (var i = elements.Length - 1; i >= 0; i--)
2120                    {
2121                        var element = elements[i];
2122                        if (element != 0)
2123                        {
2124                            CollectMatchingSequences(element, leftBound + 1, middleLinks,
                                ↪ rightLink, rightBound, ref results);
2125                        }
2126                    }
2127                }
2128                else
2129                {
2130                    for (var i = elements.Length - 1; i >= 0; i--)
2131                    {
2132                        var element = elements[i];
2133                        if (element != 0)
2134                        {
2135                            results.Add(element);
2136                        }
2137                    }
2138                }
2139            }
2140            else
2141            {
2142                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>
```

```
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The result.</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public ulong[] GetRightElements(ulong startLink, ulong rightLink)
        {
            var result = new ulong[5];
            TryStepRight(startLink, rightLink, result, 0);
            Links.Each(Constants.Any, startLink, couple =>
            {
                if (couple != startLink)
                {
                    if (TryStepRight(couple, rightLink, result, 2))
                    {
                        return false;
                    }
                }
                return true;
            });
            if (Links.GetTarget(Links.GetTarget(startLink)) == rightLink)
            {
                result[4] = startLink;
            }
            return result;
        }

        /// <summary>
        /// <para>
        /// Determines whether this instance try step right.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="startLink">
        /// <para>The start link.</para>
        /// <para></para>
        /// </param>
        /// <param name="rightLink">
        /// <para>The right link.</para>
        /// <para></para>
        /// </param>
        /// <param name="result">
        /// <para>The result.</para>
        /// <para></para>
        /// </param>
        /// <param name="offset">
        /// <para>The offset.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public bool TryStepRight(ulong startLink, ulong rightLink, ulong[] result, int offset)
        {
            var added = 0;
            Links.Each(startLink, Constants.Any, couple =>
            {
                if (couple != startLink)
                {
                    var coupleTarget = Links.GetTarget(couple);
                    if (coupleTarget == rightLink)
                    {
                        result[offset] = couple;
                        if (++added == 2)
                        {
                            return false;
                        }
                    }
                    else if (Links.GetSource(coupleTarget) == rightLink) // coupleTarget.Linker
                    ↪  == Net.And &&
                    {
                        result[offset + 1] = couple;
                        if (++added == 2)
                        {
```

```
                        return false;
                    }
                }
            }
            return true;
        });
        return added > 0;
    }

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

    /// <summary>
    /// <para>
    /// Determines whether this instance try step left.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <param name="startLink">
    /// <para>The start link.</para>
    /// <para></para>
    /// </param>
    /// <param name="leftLink">
    /// <para>The left link.</para>
    /// <para></para>
    /// </param>
    /// <param name="result">
    /// <para>The result.</para>
    /// <para></para>
    /// </param>
    /// <param name="offset">
    /// <para>The offset.</para>
    /// <para></para>
    /// </param>
    /// <returns>
    /// <para>The bool</para>
    /// <para></para>
    /// </returns>
    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    public bool TryStepLeft(ulong startLink, ulong leftLink, ulong[] result, int offset)
```

```
        {
            var added = 0;
            Links.Each(Constants.Any, startLink, couple =>
            {
                if (couple != startLink)
                {
                    var coupleSource = Links.GetSource(couple);
                    if (coupleSource == leftLink)
                    {
                        result[offset] = couple;
                        if (++added == 2)
                        {
                            return false;
                        }
                    }
                    else if (Links.GetTarget(coupleSource) == leftLink) // coupleSource.Linker
                    ↪  == Net.And &&
                    {
                        result[offset + 1] = couple;
                        if (++added == 2)
                        {
                            return false;
                        }
                    }
                }
                return true;
            });
            return added > 0;
        }

        #endregion

        #region Walkers

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

            #region Pattern Match

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

```
2414                }
2415
2416                /// <summary>
2417                /// <para>
2418                /// The pattern block.
2419                /// </para>
2420                /// <para></para>
2421                /// </summary>
2422                struct PatternBlock
2423                {
2424                    /// <summary>
2425                    /// <para>
2426                    /// The type.
2427                    /// </para>
2428                    /// <para></para>
2429                    /// </summary>
2430                    public PatternBlockType Type;
2431                    /// <summary>
2432                    /// <para>
2433                    /// The start.
2434                    /// </para>
2435                    /// <para></para>
2436                    /// </summary>
2437                    public long Start;
2438                    /// <summary>
2439                    /// <para>
2440                    /// The stop.
2441                    /// </para>
2442                    /// <para></para>
2443                    /// </summary>
2444                    public long Stop;
2445                }
2446                private readonly List<PatternBlock> _pattern;
2447                private int _patternPosition;
2448                private long _sequencePosition;
2449
2450                #endregion
2451
2452                /// <summary>
2453                /// <para>
2454                /// Initializes a new <see cref="PatternMatcher"/> instance.
2455                /// </para>
2456                /// <para></para>
2457                /// </summary>
2458                /// <param name="sequences">
2459                /// <para>A sequences.</para>
2460                /// <para></para>
2461                /// </param>
2462                /// <param name="patternSequence">
2463                /// <para>A pattern sequence.</para>
2464                /// <para></para>
2465                /// </param>
2466                /// <param name="results">
2467                /// <para>A results.</para>
2468                /// <para></para>
2469                /// </param>
2470                [MethodImpl(MethodImplOptions.AggressiveInlining)]
2471                public PatternMatcher(Sequences sequences, LinkIndex[] patternSequence,
                    ↪  HashSet<LinkIndex> results)
2472                    : base(sequences.Links.Unsync, new DefaultStack<ulong>())
2473                {
2474                    _sequences = sequences;
2475                    _patternSequence = patternSequence;
2476                    _linksInSequence = new HashSet<LinkIndex>(patternSequence.Where(x => x !=
                        ↪  _sequences.Constants.Any && x != ZeroOrMany));
2477                    _results = results;
2478                    _pattern = CreateDetailedPattern();
2479                }
2480
2481                /// <summary>
2482                /// <para>
2483                /// Determines whether this instance is element.
2484                /// </para>
2485                /// <para></para>
2486                /// </summary>
2487                /// <param name="link">
2488                /// <para>The link.</para>
2489                /// <para></para>
2490                /// </param>
```

```
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override bool IsElement(ulong link) => _linksInSequence.Contains(link) ||
        ↪  base.IsElement(link);

        /// <summary>
        /// <para>
        /// Determines whether this instance pattern match.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="sequenceToMatch">
        /// <para>The sequence to match.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public bool PatternMatch(LinkIndex sequenceToMatch)
        {
            _patternPosition = 0;
            _sequencePosition = 0;
            foreach (var part in Walk(sequenceToMatch))
            {
                if (!PatternMatchCore(part))
                {
                    break;
                }
            }
            return _patternPosition == _pattern.Count || (_patternPosition == _pattern.Count
            ↪  - 1 && _pattern[_patternPosition].Start == 0);
        }
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private List<PatternBlock> CreateDetailedPattern()
        {
            var pattern = new List<PatternBlock>();
            var patternBlock = new PatternBlock();
            for (var i = 0; i < _patternSequence.Length; i++)
            {
                if (patternBlock.Type == PatternBlockType.Undefined)
                {
                    if (_patternSequence[i] == _sequences.Constants.Any)
                    {
                        patternBlock.Type = PatternBlockType.Gap;
                        patternBlock.Start = 1;
                        patternBlock.Stop = 1;
                    }
                    else if (_patternSequence[i] == ZeroOrMany)
                    {
                        patternBlock.Type = PatternBlockType.Gap;
                        patternBlock.Start = 0;
                        patternBlock.Stop = long.MaxValue;
                    }
                    else
                    {
                        patternBlock.Type = PatternBlockType.Elements;
                        patternBlock.Start = i;
                        patternBlock.Stop = i;
                    }
                }
                else if (patternBlock.Type == PatternBlockType.Elements)
                {
                    if (_patternSequence[i] == _sequences.Constants.Any)
                    {
                        pattern.Add(patternBlock);
                        patternBlock = new PatternBlock
                        {
                            Type = PatternBlockType.Gap,
                            Start = 1,
                            Stop = 1
                        };
                    }
                    else if (_patternSequence[i] == ZeroOrMany)
                    {
```

```
                        pattern.Add(patternBlock);
                        patternBlock = new PatternBlock
                        {
                            Type = PatternBlockType.Gap,
                            Start = 0,
                            Stop = long.MaxValue
                        };
                    }
                    else
                    {
                        patternBlock.Stop = i;
                    }
                }
                else // patternBlock.Type == PatternBlockType.Gap
                {
                    if (_patternSequence[i] == _sequences.Constants.Any)
                    {
                        patternBlock.Start++;
                        if (patternBlock.Stop < patternBlock.Start)
                        {
                            patternBlock.Stop = patternBlock.Start;
                        }
                    }
                    else if (_patternSequence[i] == ZeroOrMany)
                    {
                        patternBlock.Stop = long.MaxValue;
                    }
                    else
                    {
                        pattern.Add(patternBlock);
                        patternBlock = new PatternBlock
                        {
                            Type = PatternBlockType.Elements,
                            Start = i,
                            Stop = i
                        };
                    }
                }
            }
            if (patternBlock.Type != PatternBlockType.Undefined)
            {
                pattern.Add(patternBlock);
            }
            return pattern;
        }

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

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

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

```
2647
2648            //private void GetNextPatternElement(out LinkIndex element, out long mininumGap, out
                ↪  long maximumGap)
2649            //{
2650            //    mininumGap = 0;
2651            //    maximumGap = 0;
2652            //    element = 0;
2653            //    for (; _patternPosition < _patternSequence.Length; _patternPosition++)
2654            //    {
2655            //        if (_patternSequence[_patternPosition] == Doublets.Links.Null)
2656            //            mininumGap++;
2657            //        else if (_patternSequence[_patternPosition] == ZeroOrMany)
2658            //            maximumGap = long.MaxValue;
2659            //        else
2660            //            break;
2661            //    }
2662
2663            //    if (maximumGap < mininumGap)
2664            //        maximumGap = mininumGap;
2665            //}
2666            [MethodImpl(MethodImplOptions.AggressiveInlining)]
2667            private bool PatternMatchCore(LinkIndex element)
2668            {
2669                if (_patternPosition >= _pattern.Count)
2670                {
2671                    _patternPosition = -2;
2672                    return false;
2673                }
2674                var currentPatternBlock = _pattern[_patternPosition];
2675                if (currentPatternBlock.Type == PatternBlockType.Gap)
2676                {
2677                    //var currentMatchingBlockLength = (_sequencePosition -
                        ↪  _lastMatchedBlockPosition);
2678                    if (_sequencePosition < currentPatternBlock.Start)
2679                    {
2680                        _sequencePosition++;
2681                        return true; // Двигаемся дальше
2682                    }
2683                    // Это последний блок
2684                    if (_pattern.Count == _patternPosition + 1)
2685                    {
2686                        _patternPosition++;
2687                        _sequencePosition = 0;
2688                        return false; // Полное соответствие
2689                    }
2690                    else
2691                    {
2692                        if (_sequencePosition > currentPatternBlock.Stop)
2693                        {
2694                            return false; // Соответствие невозможно
2695                        }
2696                        var nextPatternBlock = _pattern[_patternPosition + 1];
2697                        if (_patternSequence[nextPatternBlock.Start] == element)
2698                        {
2699                            if (nextPatternBlock.Start < nextPatternBlock.Stop)
2700                            {
2701                                _patternPosition++;
2702                                _sequencePosition = 1;
2703                            }
2704                            else
2705                            {
2706                                _patternPosition += 2;
2707                                _sequencePosition = 0;
2708                            }
2709                        }
2710                    }
2711                }
2712                else // currentPatternBlock.Type == PatternBlockType.Elements
2713                {
2714                    var patternElementPosition = currentPatternBlock.Start + _sequencePosition;
2715                    if (_patternSequence[patternElementPosition] != element)
2716                    {
2717                        return false; // Соответствие невозможно
2718                    }
2719                    if (patternElementPosition == currentPatternBlock.Stop)
2720                    {
2721                        _patternPosition++;
2722                        _sequencePosition = 0;
2723                    }
```

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

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

        #endregion
    }
}
```

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

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Runtime.CompilerServices;
using Platform.Collections;
using Platform.Collections.Lists;
using Platform.Collections.Stacks;
using Platform.Threading.Synchronization;
using Platform.Data.Doublets.Sequences.Walkers;
```

```
using LinkIndex = System.UInt64;

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

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

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

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

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

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

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

        /// <summary>
        /// <para>
        /// Determines whether this instance is sequence.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="sequence">
        /// <para>The sequence.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public bool IsSequence(LinkIndex sequence)
        {
            return _sync.ExecuteReadOperation(() =>
            {
                if (Options.UseSequenceMarker)
                {
                    return Options.MarkedSequenceMatcher.IsMatched(sequence);
                }
                return !Links.Unsync.IsPartialPoint(sequence);
            });
        }
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private LinkIndex GetSequenceByElements(LinkIndex sequence)
        {
            if (Options.UseSequenceMarker)
            {
                return Links.SearchOrDefault(Options.SequenceMarkerLink, sequence);
            }
            return sequence;
        }
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private LinkIndex GetSequenceElements(LinkIndex sequence)
        {
            if (Options.UseSequenceMarker)
            {
                var linkContents = new Link<ulong>(Links.GetLink(sequence));
```

```
                if (linkContents.Source == Options.SequenceMarkerLink)
                {
                    return linkContents.Target;
                }
                if (linkContents.Target == Options.SequenceMarkerLink)
                {
                    return linkContents.Source;
                }
            }
            return sequence;
        }

        #region Count

        /// <summary>
        /// <para>
        /// Counts the restrictions.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="restrictions">
        /// <para>The restrictions.</para>
        /// <para></para>
        /// </param>
        /// <exception cref="NotImplementedException">
        /// <para></para>
        /// <para></para>
        /// </exception>
        /// <returns>
        /// <para>The link index</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public LinkIndex Count(IList<LinkIndex> restrictions)
        {
            if (restrictions.IsNullOrEmpty())
            {
                return Links.Count(Constants.Any, Options.SequenceMarkerLink, Constants.Any);
            }
            if (restrictions.Count == 1) // Первая связь это адрес
            {
                var sequenceIndex = restrictions[0];
                if (sequenceIndex == Constants.Null)
                {
                    return 0;
                }
                if (sequenceIndex == Constants.Any)
                {
                    return Count(null);
                }
                if (Options.UseSequenceMarker)
                {
                    return Links.Count(Constants.Any, Options.SequenceMarkerLink, sequenceIndex);
                }
                return Links.Exists(sequenceIndex) ? 1UL : 0;
            }
            throw new NotImplementedException();
        }
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private LinkIndex CountUsages(params LinkIndex[] restrictions)
        {
            if (restrictions.Length == 0)
            {
                return 0;
            }
            if (restrictions.Length == 1) // Первая связь это адрес
            {
                if (restrictions[0] == Constants.Null)
                {
                    return 0;
                }
                var any = Constants.Any;
                if (Options.UseSequenceMarker)
                {
                    var elementsLink = GetSequenceElements(restrictions[0]);
                    var sequenceLink = GetSequenceByElements(elementsLink);
                    if (sequenceLink != Constants.Null)
                    {
```

```
                        return Links.Count(any, sequenceLink) + Links.Count(any, elementsLink) -
                        ↪ 1;
                    }
                    return Links.Count(any, elementsLink);
                }
                return Links.Count(any, restrictions[0]);
            }
            throw new NotImplementedException();
        }

        #endregion

        #region Create

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

        #endregion

        #region Each

        /// <summary>
```

```
313         /// <para>
314         /// Eaches the sequence.
315         /// </para>
316         /// <para></para>
317         /// </summary>
318         /// <param name="sequence">
319         /// <para>The sequence.</para>
320         /// <para></para>
321         /// </param>
322         /// <returns>
323         /// <para>The results.</para>
324         /// <para></para>
325         /// </returns>
326         [MethodImpl(MethodImplOptions.AggressiveInlining)]
327         public List<LinkIndex> Each(IList<LinkIndex> sequence)
328         {
329             var results = new List<LinkIndex>();
330             var filler = new ListFiller<LinkIndex, LinkIndex>(results, Constants.Continue);
331             Each(filler.AddFirstAndReturnConstant, sequence);
332             return results;
333         }
334 
335         /// <summary>
336         /// <para>
337         /// Eaches the handler.
338         /// </para>
339         /// <para></para>
340         /// </summary>
341         /// <param name="handler">
342         /// <para>The handler.</para>
343         /// <para></para>
344         /// </param>
345         /// <param name="restrictions">
346         /// <para>The restrictions.</para>
347         /// <para></para>
348         /// </param>
349         /// <exception cref="NotImplementedException">
350         /// <para></para>
351         /// <para></para>
352         /// </exception>
353         /// <returns>
354         /// <para>The link index</para>
355         /// <para></para>
356         /// </returns>
357         [MethodImpl(MethodImplOptions.AggressiveInlining)]
358         public LinkIndex Each(Func<IList<LinkIndex>, LinkIndex> handler, IList<LinkIndex>
            ↪ restrictions)
359         {
360             return _sync.ExecuteReadOperation(() =>
361             {
362                 if (restrictions.IsNullOrEmpty())
363                 {
364                     return Constants.Continue;
365                 }
366                 Links.EnsureInnerReferenceExists(restrictions, nameof(restrictions));
367                 if (restrictions.Count == 1)
368                 {
369                     var link = restrictions[0];
370                     var any = Constants.Any;
371                     if (link == any)
372                     {
373                         if (Options.UseSequenceMarker)
374                         {
375                             return Links.Unsync.Each(handler, new Link<LinkIndex>(any,
                                ↪ Options.SequenceMarkerLink, any));
376                         }
377                         else
378                         {
379                             return Links.Unsync.Each(handler, new Link<LinkIndex>(any, any,
                                ↪ any));
380                         }
381                     }
382                     if (Options.UseSequenceMarker)
383                     {
384                         var sequenceLinkValues = Links.Unsync.GetLink(link);
385                         if (sequenceLinkValues[Constants.SourcePart] ==
                            ↪ Options.SequenceMarkerLink)
386                         {
```

```
                            link = sequenceLinkValues[Constants.TargetPart];
                        }
                    }
                    var sequence = Options.Walker.Walk(link).ToArray().ShiftRight();
                    sequence[0] = link;
                    return handler(sequence);
                }
                else if (restrictions.Count == 2)
                {
                    throw new NotImplementedException();
                }
                else if (restrictions.Count == 3)
                {
                    return Links.Unsync.Each(handler, restrictions);
                }
                else
                {
                    var sequence = restrictions.SkipFirst();
                    if (Options.UseIndex && !Options.Index.MightContain(sequence))
                    {
                        return Constants.Break;
                    }
                    return EachCore(handler, sequence);
                }
            });
        }
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private LinkIndex EachCore(Func<IList<LinkIndex>, LinkIndex> handler, IList<LinkIndex>
        ↪  values)
        {
            var matcher = new Matcher(this, values, new HashSet<LinkIndex>(), handler);
            // TODO: Find out why matcher.HandleFullMatched executed twice for the same sequence
            ↪  Id.
            Func<IList<LinkIndex>, LinkIndex> innerHandler = Options.UseSequenceMarker ?
            ↪  (Func<IList<LinkIndex>, LinkIndex>)matcher.HandleFullMatchedSequence :
            ↪  matcher.HandleFullMatched;
            //if (sequence.Length >= 2)
            if (StepRight(innerHandler, values[0], values[1]) != Constants.Continue)
            {
                return Constants.Break;
            }
            var last = values.Count - 2;
            for (var i = 1; i < last; i++)
            {
                if (PartialStepRight(innerHandler, values[i], values[i + 1]) !=
                ↪  Constants.Continue)
                {
                    return Constants.Break;
                }
            }
            if (values.Count >= 3)
            {
                if (StepLeft(innerHandler, values[values.Count - 2], values[values.Count - 1])
                ↪  != Constants.Continue)
                {
                    return Constants.Break;
                }
            }
            return Constants.Continue;
        }
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private LinkIndex PartialStepRight(Func<IList<LinkIndex>, LinkIndex> handler, LinkIndex
        ↪  left, LinkIndex right)
        {
            return Links.Unsync.Each(doublet =>
            {
                var doubletIndex = doublet[Constants.IndexPart];
                if (StepRight(handler, doubletIndex, right) != Constants.Continue)
                {
                    return Constants.Break;
                }
                if (left != doubletIndex)
                {
                    return PartialStepRight(handler, doubletIndex, right);
                }
                return Constants.Continue;
            }, new Link<LinkIndex>(Constants.Any, Constants.Any, left));
        }
```

```
458             [MethodImpl(MethodImplOptions.AggressiveInlining)]
459             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));
460             [MethodImpl(MethodImplOptions.AggressiveInlining)]
461             private LinkIndex TryStepRightUp(Func<IList<LinkIndex>, LinkIndex> handler, LinkIndex
                    right, LinkIndex stepFrom)
462             {
463                 var upStep = stepFrom;
464                 var firstSource = Links.Unsync.GetTarget(upStep);
465                 while (firstSource != right && firstSource != upStep)
466                 {
467                     upStep = firstSource;
468                     firstSource = Links.Unsync.GetSource(upStep);
469                 }
470                 if (firstSource == right)
471                 {
472                     return handler(new LinkAddress<LinkIndex>(stepFrom));
473                 }
474                 return Constants.Continue;
475             }
476             [MethodImpl(MethodImplOptions.AggressiveInlining)]
477             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));
478             [MethodImpl(MethodImplOptions.AggressiveInlining)]
479             private LinkIndex TryStepLeftUp(Func<IList<LinkIndex>, LinkIndex> handler, LinkIndex
                    left, LinkIndex stepFrom)
480             {
481                 var upStep = stepFrom;
482                 var firstTarget = Links.Unsync.GetSource(upStep);
483                 while (firstTarget != left && firstTarget != upStep)
484                 {
485                     upStep = firstTarget;
486                     firstTarget = Links.Unsync.GetTarget(upStep);
487                 }
488                 if (firstTarget == left)
489                 {
490                     return handler(new LinkAddress<LinkIndex>(stepFrom));
491                 }
492                 return Constants.Continue;
493             }
494
495             #endregion
496
497             #region Update
498
499             /// <summary>
500             /// <para>
501             /// Updates the restrictions.
502             /// </para>
503             /// <para></para>
504             /// </summary>
505             /// <param name="restrictions">
506             /// <para>The restrictions.</para>
507             /// <para></para>
508             /// </param>
509             /// <param name="substitution">
510             /// <para>The substitution.</para>
511             /// <para></para>
512             /// </param>
513             /// <returns>
514             /// <para>The link index</para>
515             /// <para></para>
516             /// </returns>
517             [MethodImpl(MethodImplOptions.AggressiveInlining)]
518             public LinkIndex Update(IList<LinkIndex> restrictions, IList<LinkIndex> substitution)
519             {
520                 var sequence = restrictions.SkipFirst();
521                 var newSequence = substitution.SkipFirst();
522                 if (sequence.IsNullOrEmpty() && newSequence.IsNullOrEmpty())
523                 {
524                     return Constants.Null;
525                 }
526                 if (sequence.IsNullOrEmpty())
527                 {
```

```
528                 return Create(substitution);
529             }
530             if (newSequence.IsNullOrEmpty())
531             {
532                 Delete(restrictions);
533                 return Constants.Null;
534             }
535             return _sync.ExecuteWriteOperation((Func<ulong>)(() =>
536             {
537                 ILinksExtensions.EnsureLinkIsAnyOrExists<ulong>(Links, (IList<ulong>)sequence);
538                 Links.EnsureLinkExists(newSequence);
539                 return UpdateCore(sequence, newSequence);
540             }));
541         }
542         [MethodImpl(MethodImplOptions.AggressiveInlining)]
543         private LinkIndex UpdateCore(IList<LinkIndex> sequence, IList<LinkIndex> newSequence)
544         {
545             LinkIndex bestVariant;
546             if (Options.EnforceSingleSequenceVersionOnWriteBasedOnNew &&
                ↪  !sequence.EqualTo(newSequence))
547             {
548                 bestVariant = CompactCore(newSequence);
549             }
550             else
551             {
552                 bestVariant = CreateCore(newSequence);
553             }
554             // TODO: Check all options only ones before loop execution
555             // Возможно нужно две версии Each, возвращающий фактические последовательности и с
                ↪  маркером,
556             // или возможно даже возвращать и тот и тот вариант. С другой стороны все варианты
                ↪  можно получить имея только фактические последовательности.
557             foreach (var variant in Each(sequence))
558             {
559                 if (variant != bestVariant)
560                 {
561                     UpdateOneCore(variant, bestVariant);
562                 }
563             }
564             return bestVariant;
565         }
566         [MethodImpl(MethodImplOptions.AggressiveInlining)]
567         private void UpdateOneCore(LinkIndex sequence, LinkIndex newSequence)
568         {
569             if (Options.UseGarbageCollection)
570             {
571                 var sequenceElements = GetSequenceElements(sequence);
572                 var sequenceElementsContents = new Link<ulong>(Links.GetLink(sequenceElements));
573                 var sequenceLink = GetSequenceByElements(sequenceElements);
574                 var newSequenceElements = GetSequenceElements(newSequence);
575                 var newSequenceLink = GetSequenceByElements(newSequenceElements);
576                 if (Options.UseCascadeUpdate || CountUsages(sequence) == 0)
577                 {
578                     if (sequenceLink != Constants.Null)
579                     {
580                         Links.Unsync.MergeAndDelete(sequenceLink, newSequenceLink);
581                     }
582                     Links.Unsync.MergeAndDelete(sequenceElements, newSequenceElements);
583                 }
584                 ClearGarbage(sequenceElementsContents.Source);
585                 ClearGarbage(sequenceElementsContents.Target);
586             }
587             else
588             {
589                 if (Options.UseSequenceMarker)
590                 {
591                     var sequenceElements = GetSequenceElements(sequence);
592                     var sequenceLink = GetSequenceByElements(sequenceElements);
593                     var newSequenceElements = GetSequenceElements(newSequence);
594                     var newSequenceLink = GetSequenceByElements(newSequenceElements);
595                     if (Options.UseCascadeUpdate || CountUsages(sequence) == 0)
596                     {
597                         if (sequenceLink != Constants.Null)
598                         {
599                             Links.Unsync.MergeAndDelete(sequenceLink, newSequenceLink);
600                         }
601                         Links.Unsync.MergeAndDelete(sequenceElements, newSequenceElements);
602                     }
```

```
                }
                else
                {
                    if (Options.UseCascadeUpdate || CountUsages(sequence) == 0)
                    {
                        Links.Unsync.MergeAndDelete(sequence, newSequence);
                    }
                }
            }
        }

        #endregion

        #region Delete

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

```
            }
        }
    }

    #endregion

    #region Compactification

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

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

    #endregion

    #region Garbage Collection
    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    private bool IsGarbage(LinkIndex link) => link != Options.SequenceMarkerLink &&
    ↪  !Links.Unsync.IsPartialPoint(link) && Links.Count(Constants.Any, link) == 0;
    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    private void ClearGarbage(LinkIndex link)
    {
        if (IsGarbage(link))
        {
            var contents = new Link<ulong>(Links.GetLink(link));
            Links.Unsync.Delete(link);
            ClearGarbage(contents.Source);
            ClearGarbage(contents.Target);
        }
    }

    #endregion

    #region Walkers

    /// <summary>
    /// <para>
    /// Determines whether this instance each part.
    /// </para>
    /// <para></para>
```

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

```
835                 _sequences = sequences;
836                 _patternSequence = patternSequence;
837                 _linksInSequence = new HashSet<LinkIndex>(patternSequence.Where(x => x !=
                    ↪  _links.Constants.Any && x != ZeroOrMany));
838                 _results = results;
839                 _stopableHandler = stopableHandler;
840                 _readAsElements = readAsElements;
841             }
842 
843             /// <summary>
844             /// <para>
845             /// Determines whether this instance is element.
846             /// </para>
847             /// <para></para>
848             /// </summary>
849             /// <param name="link">
850             /// <para>The link.</para>
851             /// <para></para>
852             /// </param>
853             /// <returns>
854             /// <para>The bool</para>
855             /// <para></para>
856             /// </returns>
857             [MethodImpl(MethodImplOptions.AggressiveInlining)]
858             protected override bool IsElement(LinkIndex link) => base.IsElement(link) ||
                ↪  (_readAsElements != null && _readAsElements.Contains(link)) ||
                ↪  _linksInSequence.Contains(link);
859 
860             /// <summary>
861             /// <para>
862             /// Determines whether this instance full match.
863             /// </para>
864             /// <para></para>
865             /// </summary>
866             /// <param name="sequenceToMatch">
867             /// <para>The sequence to match.</para>
868             /// <para></para>
869             /// </param>
870             /// <returns>
871             /// <para>The bool</para>
872             /// <para></para>
873             /// </returns>
874             [MethodImpl(MethodImplOptions.AggressiveInlining)]
875             public bool FullMatch(LinkIndex sequenceToMatch)
876             {
877                 _filterPosition = 0;
878                 foreach (var part in Walk(sequenceToMatch))
879                 {
880                     if (!FullMatchCore(part))
881                     {
882                         break;
883                     }
884                 }
885                 return _filterPosition == _patternSequence.Count;
886             }
887             [MethodImpl(MethodImplOptions.AggressiveInlining)]
888             private bool FullMatchCore(LinkIndex element)
889             {
890                 if (_filterPosition == _patternSequence.Count)
891                 {
892                     _filterPosition = -2; // Длиннее чем нужно
893                     return false;
894                 }
895                 if (_patternSequence[_filterPosition] != _links.Constants.Any
896                  && element != _patternSequence[_filterPosition])
897                 {
898                     _filterPosition = -1;
899                     return false; // Начинается/Продолжается иначе
900                 }
901                 _filterPosition++;
902                 return true;
903             }
904 
905             /// <summary>
906             /// <para>
907             /// Adds the full matched to results using the specified restrictions.
908             /// </para>
909             /// <para></para>
910             /// </summary>
```

```
        /// <param name="restrictions">
        /// <para>The restrictions.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void AddFullMatchedToResults(IList<LinkIndex> restrictions)
        {
            var sequenceToMatch = restrictions[_links.Constants.IndexPart];
            if (FullMatch(sequenceToMatch))
            {
                _results.Add(sequenceToMatch);
            }
        }

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

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

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

```
                }
            }
            return _filterPosition == _patternSequence.Count - 1;
        }
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private bool PartialMatchCore(LinkIndex element)
        {
            if (_filterPosition == (_patternSequence.Count - 1))
            {
                return false; // Нашлось
            }
            if (_filterPosition >= 0)
            {
                if (element == _patternSequence[_filterPosition + 1])
                {
                    _filterPosition++;
                }
                else
                {
                    _filterPosition = -1;
                }
            }
            if (_filterPosition < 0)
            {
                if (element == _patternSequence[0])
                {
                    _filterPosition = 0;
                }
            }
            return true; // Ищем дальше
        }

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

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

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

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

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

        #endregion
    }
}
```

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

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

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

namespace Platform.Data.Doublets.Sequences
{
    /// <summary>
    /// <para>
    /// Represents the sequences extensions.
    /// </para>
    /// <para></para>
    /// </summary>
    public static class SequencesExtensions
    {
        /// <summary>
        /// <para>
        /// Creates the sequences.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <typeparam name="TLink">
        /// <para>The link.</para>
        /// <para></para>
        /// </typeparam>
        /// <param name="sequences">
```

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

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

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

```
20      /// Represents the sequences options.
21      /// </para>
22      /// <para></para>
23      /// </summary>
24      public class SequencesOptions<TLink> // TODO: To use type parameter <TLink> the
        ↪  ILinks<TLink> must contain GetConstants function.
25      {
26          private static readonly EqualityComparer<TLink> _equalityComparer =
            ↪  EqualityComparer<TLink>.Default;
27
28          /// <summary>
29          /// <para>
30          /// Gets or sets the sequence marker link value.
31          /// </para>
32          /// <para></para>
33          /// </summary>
34          public TLink SequenceMarkerLink
35          {
36              [MethodImpl(MethodImplOptions.AggressiveInlining)]
37              get;
38              [MethodImpl(MethodImplOptions.AggressiveInlining)]
39              set;
40          }
41
42          /// <summary>
43          /// <para>
44          /// Gets or sets the use cascade update value.
45          /// </para>
46          /// <para></para>
47          /// </summary>
48          public bool UseCascadeUpdate
49          {
50              [MethodImpl(MethodImplOptions.AggressiveInlining)]
51              get;
52              [MethodImpl(MethodImplOptions.AggressiveInlining)]
53              set;
54          }
55
56          /// <summary>
57          /// <para>
58          /// Gets or sets the use cascade delete value.
59          /// </para>
60          /// <para></para>
61          /// </summary>
62          public bool UseCascadeDelete
63          {
64              [MethodImpl(MethodImplOptions.AggressiveInlining)]
65              get;
66              [MethodImpl(MethodImplOptions.AggressiveInlining)]
67              set;
68          }
69
70          /// <summary>
71          /// <para>
72          /// Gets or sets the use index value.
73          /// </para>
74          /// <para></para>
75          /// </summary>
76          public bool UseIndex
77          {
78              [MethodImpl(MethodImplOptions.AggressiveInlining)]
79              get;
80              [MethodImpl(MethodImplOptions.AggressiveInlining)]
81              set;
82          } // TODO: Update Index on sequence update/delete.
83
84          /// <summary>
85          /// <para>
86          /// Gets or sets the use sequence marker value.
87          /// </para>
88          /// <para></para>
89          /// </summary>
90          public bool UseSequenceMarker
91          {
92              [MethodImpl(MethodImplOptions.AggressiveInlining)]
93              get;
94              [MethodImpl(MethodImplOptions.AggressiveInlining)]
95              set;
96          }
```

```
97
98          /// <summary>
99          /// <para>
100         /// Gets or sets the use compression value.
101         /// </para>
102         /// <para></para>
103         /// </summary>
104         public bool UseCompression
105         {
106             [MethodImpl(MethodImplOptions.AggressiveInlining)]
107             get;
108             [MethodImpl(MethodImplOptions.AggressiveInlining)]
109             set;
110         }
111
112         /// <summary>
113         /// <para>
114         /// Gets or sets the use garbage collection value.
115         /// </para>
116         /// <para></para>
117         /// </summary>
118         public bool UseGarbageCollection
119         {
120             [MethodImpl(MethodImplOptions.AggressiveInlining)]
121             get;
122             [MethodImpl(MethodImplOptions.AggressiveInlining)]
123             set;
124         }
125
126         /// <summary>
127         /// <para>
128         /// Gets or sets the enforce single sequence version on write based on existing value.
129         /// </para>
130         /// <para></para>
131         /// </summary>
132         public bool EnforceSingleSequenceVersionOnWriteBasedOnExisting
133         {
134             [MethodImpl(MethodImplOptions.AggressiveInlining)]
135             get;
136             [MethodImpl(MethodImplOptions.AggressiveInlining)]
137             set;
138         }
139
140         /// <summary>
141         /// <para>
142         /// Gets or sets the enforce single sequence version on write based on new value.
143         /// </para>
144         /// <para></para>
145         /// </summary>
146         public bool EnforceSingleSequenceVersionOnWriteBasedOnNew
147         {
148             [MethodImpl(MethodImplOptions.AggressiveInlining)]
149             get;
150             [MethodImpl(MethodImplOptions.AggressiveInlining)]
151             set;
152         }
153
154         /// <summary>
155         /// <para>
156         /// Gets or sets the marked sequence matcher value.
157         /// </para>
158         /// <para></para>
159         /// </summary>
160         public MarkedSequenceCriterionMatcher<TLink> MarkedSequenceMatcher
161         {
162             [MethodImpl(MethodImplOptions.AggressiveInlining)]
163             get;
164             [MethodImpl(MethodImplOptions.AggressiveInlining)]
165             set;
166         }
167
168         /// <summary>
169         /// <para>
170         /// Gets or sets the links to sequence converter value.
171         /// </para>
172         /// <para></para>
173         /// </summary>
174         public IConverter<IList<TLink>, TLink> LinksToSequenceConverter
175         {
```

```
176             [MethodImpl(MethodImplOptions.AggressiveInlining)]
177             get;
178             [MethodImpl(MethodImplOptions.AggressiveInlining)]
179             set;
180         }
181 
182         /// <summary>
183         /// <para>
184         /// Gets or sets the index value.
185         /// </para>
186         /// <para></para>
187         /// </summary>
188         public ISequenceIndex<TLink> Index
189         {
190             [MethodImpl(MethodImplOptions.AggressiveInlining)]
191             get;
192             [MethodImpl(MethodImplOptions.AggressiveInlining)]
193             set;
194         }
195 
196         /// <summary>
197         /// <para>
198         /// Gets or sets the walker value.
199         /// </para>
200         /// <para></para>
201         /// </summary>
202         public ISequenceWalker<TLink> Walker
203         {
204             [MethodImpl(MethodImplOptions.AggressiveInlining)]
205             get;
206             [MethodImpl(MethodImplOptions.AggressiveInlining)]
207             set;
208         }
209 
210         /// <summary>
211         /// <para>
212         /// Gets or sets the read full sequence value.
213         /// </para>
214         /// <para></para>
215         /// </summary>
216         public bool ReadFullSequence
217         {
218             [MethodImpl(MethodImplOptions.AggressiveInlining)]
219             get;
220             [MethodImpl(MethodImplOptions.AggressiveInlining)]
221             set;
222         }
223 
224         // TODO: Реализовать компактификацию при чтении
225         //public bool EnforceSingleSequenceVersionOnRead { get; set; }
226         //public bool UseRequestMarker { get; set; }
227         //public bool StoreRequestResults { get; set; }
228 
229         /// <summary>
230         /// <para>
231         /// Inits the options using the specified links.
232         /// </para>
233         /// <para></para>
234         /// </summary>
235         /// <param name="links">
236         /// <para>The links.</para>
237         /// <para></para>
238         /// </param>
239         /// <exception cref="InvalidOperationException">
240         /// <para>Cannot recreate sequence marker link.</para>
241         /// <para></para>
242         /// </exception>
243         [MethodImpl(MethodImplOptions.AggressiveInlining)]
244         public void InitOptions(ISynchronizedLinks<TLink> links)
245         {
246             if (UseSequenceMarker)
247             {
248                 if (_equalityComparer.Equals(SequenceMarkerLink, links.Constants.Null))
249                 {
250                     SequenceMarkerLink = links.CreatePoint();
251                 }
252                 else
253                 {
254                     if (!links.Exists(SequenceMarkerLink))
```

```
255                     {
256                         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<TLink>(links,
                        ↪ SequenceMarkerLink);
266                 }
267             }
268             var balancedVariantConverter = new BalancedVariantConverter<TLink>(links);
269             if (UseCompression)
270             {
271                 if (LinksToSequenceConverter == null)
272                 {
273                     ICounter<TLink, TLink> totalSequenceSymbolFrequencyCounter;
274                     if (UseSequenceMarker)
275                     {
276                         totalSequenceSymbolFrequencyCounter = new
                            ↪ TotalMarkedSequenceSymbolFrequencyCounter<TLink>(links,
                            ↪ MarkedSequenceMatcher);
277                     }
278                     else
279                     {
280                         totalSequenceSymbolFrequencyCounter = new
                            ↪ TotalSequenceSymbolFrequencyCounter<TLink>(links);
281                     }
282                     var doubletFrequenciesCache = new LinkFrequenciesCache<TLink>(links,
                        ↪ totalSequenceSymbolFrequencyCounter);
283                     var compressingConverter = new CompressingConverter<TLink>(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<TLink>(links);
297             }
298             if (Walker == null)
299             {
300                 Walker = new RightSequenceWalker<TLink>(links, new DefaultStack<TLink>());
301             }
302         }
303 
304         /// <summary>
305         /// <para>
306         /// Validates the options.
307         /// </para>
308         /// <para></para>
309         /// </summary>
310         /// <exception cref="NotSupportedException">
311         /// <para>To use garbage collection UseSequenceMarker option must be on.</para>
312         /// <para></para>
313         /// </exception>
314         [MethodImpl(MethodImplOptions.AggressiveInlining)]
315         public void ValidateOptions()
316         {
317             if (UseGarbageCollection && !UseSequenceMarker)
318             {
319                 throw new NotSupportedException("To use garbage collection UseSequenceMarker
                    ↪ option must be on.");
320             }
321         }
322     }
323 }
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
    /// </summary>
    /// <seealso cref="LinksOperatorBase{TLink}"/>
    /// <seealso cref="IConverter{char, TLink}"/>
    public class CharToUnicodeSymbolConverter<TLink> : LinksOperatorBase<TLink>,
    ↪  IConverter<char, TLink>
    {
        private static readonly UncheckedConverter<char, TLink> _charToAddressConverter =
        ↪  UncheckedConverter<char, TLink>.Default;
        private readonly IConverter<TLink> _addressToNumberConverter;
        private readonly TLink _unicodeSymbolMarker;

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

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

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

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

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

namespace Platform.Data.Doublets.Unicode
{
    /// <summary>
    /// <para>
    /// Represents the string to unicode sequence converter.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="LinksOperatorBase{TLink}"/>
    /// <seealso cref="IConverter{string, TLink}"/>
```

```
18      public class StringToUnicodeSequenceConverter<TLink> : LinksOperatorBase<TLink>,
        ↪  IConverter<string, TLink>
19      {
20          private readonly IConverter<string, IList<TLink>> _stringToUnicodeSymbolListConverter;
21          private readonly IConverter<IList<TLink>, TLink> _unicodeSymbolListToSequenceConverter;
22
23          /// <summary>
24          /// <para>
25          /// Initializes a new <see cref="StringToUnicodeSequenceConverter"/> instance.
26          /// </para>
27          /// <para></para>
28          /// </summary>
29          /// <param name="links">
30          /// <para>A links.</para>
31          /// <para></para>
32          /// </param>
33          /// <param name="stringToUnicodeSymbolListConverter">
34          /// <para>A string to unicode symbol list converter.</para>
35          /// <para></para>
36          /// </param>
37          /// <param name="unicodeSymbolListToSequenceConverter">
38          /// <para>A unicode symbol list to sequence converter.</para>
39          /// <para></para>
40          /// </param>
41          [MethodImpl(MethodImplOptions.AggressiveInlining)]
42          public StringToUnicodeSequenceConverter(ILinks<TLink> links, IConverter<string,
            ↪  IList<TLink>> stringToUnicodeSymbolListConverter, IConverter<IList<TLink>, TLink>
            ↪  unicodeSymbolListToSequenceConverter) : base(links)
43          {
44              _stringToUnicodeSymbolListConverter = stringToUnicodeSymbolListConverter;
45              _unicodeSymbolListToSequenceConverter = unicodeSymbolListToSequenceConverter;
46          }
47
48          /// <summary>
49          /// <para>
50          /// Initializes a new <see cref="StringToUnicodeSequenceConverter"/> instance.
51          /// </para>
52          /// <para></para>
53          /// </summary>
54          /// <param name="links">
55          /// <para>A links.</para>
56          /// <para></para>
57          /// </param>
58          /// <param name="stringToUnicodeSymbolListConverter">
59          /// <para>A string to unicode symbol list converter.</para>
60          /// <para></para>
61          /// </param>
62          /// <param name="index">
63          /// <para>A index.</para>
64          /// <para></para>
65          /// </param>
66          /// <param name="listToSequenceLinkConverter">
67          /// <para>A list to sequence link converter.</para>
68          /// <para></para>
69          /// </param>
70          /// <param name="unicodeSequenceMarker">
71          /// <para>A unicode sequence marker.</para>
72          /// <para></para>
73          /// </param>
74          [MethodImpl(MethodImplOptions.AggressiveInlining)]
75          public StringToUnicodeSequenceConverter(ILinks<TLink> links, IConverter<string,
            ↪  IList<TLink>> stringToUnicodeSymbolListConverter, ISequenceIndex<TLink> index,
            ↪  IConverter<IList<TLink>, TLink> listToSequenceLinkConverter, TLink
            ↪  unicodeSequenceMarker)
76              : this(links, stringToUnicodeSymbolListConverter, new
                ↪  UnicodeSymbolsListToUnicodeSequenceConverter<TLink>(links, index,
                ↪  listToSequenceLinkConverter, unicodeSequenceMarker)) { }
77
78          /// <summary>
79          /// <para>
80          /// Initializes a new <see cref="StringToUnicodeSequenceConverter"/> instance.
81          /// </para>
82          /// <para></para>
83          /// </summary>
84          /// <param name="links">
85          /// <para>A links.</para>
86          /// <para></para>
87          /// </param>
```

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

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="StringToUnicodeSequenceConverter"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="links">
        /// <para>A links.</para>
        /// <para></para>
        /// </param>
        /// <param name="charToUnicodeSymbolConverter">
        /// <para>A char to unicode symbol converter.</para>
        /// <para></para>
        /// </param>
        /// <param name="listToSequenceLinkConverter">
        /// <para>A list to sequence link converter.</para>
        /// <para></para>
        /// </param>
        /// <param name="unicodeSequenceMarker">
        /// <para>A unicode sequence marker.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public StringToUnicodeSequenceConverter(ILinks<TLink> links, IConverter<char, TLink>
        ↪  charToUnicodeSymbolConverter, IConverter<IList<TLink>, TLink>
        ↪  listToSequenceLinkConverter, TLink unicodeSequenceMarker)
            : this(links, charToUnicodeSymbolConverter, new Unindex<TLink>(),
            ↪  listToSequenceLinkConverter, unicodeSequenceMarker) { }

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

```
157         public StringToUnicodeSequenceConverter(ILinks<TLink> links, IConverter<string,
            ↪  IList<TLink>> stringToUnicodeSymbolListConverter, IConverter<IList<TLink>, TLink>
            ↪  listToSequenceLinkConverter, TLink unicodeSequenceMarker)
158             : this(links, stringToUnicodeSymbolListConverter, new Unindex<TLink>(),
                ↪  listToSequenceLinkConverter, unicodeSequenceMarker) { }
159 
160         /// <summary>
161         /// <para>
162         /// Converts the source.
163         /// </para>
164         /// <para></para>
165         /// </summary>
166         /// <param name="source">
167         /// <para>The source.</para>
168         /// <para></para>
169         /// </param>
170         /// <returns>
171         /// <para>The link</para>
172         /// <para></para>
173         /// </returns>
174         [MethodImpl(MethodImplOptions.AggressiveInlining)]
175         public TLink Convert(string source)
176         {
177             var elements = _stringToUnicodeSymbolListConverter.Convert(source);
178             return _unicodeSymbolListToSequenceConverter.Convert(elements);
179         }
180     }
181 }
```

1.51 ./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{TLink}}"/>
16     public class StringToUnicodeSymbolsListConverter<TLink> : IConverter<string, IList<TLink>>
17     {
18         private readonly IConverter<char, TLink> _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, TLink>
            ↪  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>
41         /// <para></para>
42         /// </param>
43         /// <returns>
44         /// <para>The elements.</para>
45         /// <para></para>
46         /// </returns>
```

```
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public IList<TLink> Convert(string source)
        {
            var elements = new TLink[source.Length];
            for (var i = 0; i < elements.Length; i++)
            {
                elements[i] = _charToUnicodeSymbolConverter.Convert(source[i]);
            }
            return elements;
        }
    }
}
```

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

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

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

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

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

        /// <summary>
        /// <para>
        /// Inits the new using the specified links.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="links">
        /// <para>The links.</para>
        /// <para></para>
```

```
        /// </param>
        /// <returns>
        /// <para>The map.</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static UnicodeMap InitNew(ILinks<ulong> links)
        {
            var map = new UnicodeMap(links);
            map.Init();
            return map;
        }

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

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

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

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

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

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

        /// <summary>
        /// <para>
        /// Creates the links to string using the specified links list.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="linksList">
        /// <para>The links list.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The string</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static string FromLinksToString(IList<ulong> linksList)
        {
            var sb = new StringBuilder();
            for (int i = 0; i < linksList.Count; i++)
            {
                sb.Append(FromLinkToChar(linksList[i]));
            }
            return sb.ToString();
        }

        /// <summary>
        /// <para>
        /// Creates the sequence link to string using the specified link.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="link">
        /// <para>The link.</para>
        /// <para></para>
        /// </param>
        /// <param name="links">
        /// <para>The links.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The string</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static string FromSequenceLinkToString(ulong link, ILinks<ulong> links)
        {
            var sb = new StringBuilder();
```

```
            if (links.Exists(link))
            {
                StopableSequenceWalker.WalkRight(link, links.GetSource, links.GetTarget,
                    x => x <= MapSize || links.GetSource(x) == x || links.GetTarget(x) == x,
                    element =>
                    {
                        sb.Append(FromLinkToChar(element));
                        return true;
                    });
            }
            return sb.ToString();
        }

        /// <summary>
        /// <para>
        /// Creates the chars to link array using the specified chars.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="chars">
        /// <para>The chars.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The ulong array</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static ulong[] FromCharsToLinkArray(char[] chars) => FromCharsToLinkArray(chars,
        chars.Length);

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

        /// <summary>
        /// <para>
        /// Creates the string to link array using the specified sequence.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="sequence">
        /// <para>The sequence.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The links sequence.</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static ulong[] FromStringToLinkArray(string sequence)
        {
```

```
296             // char array to ulong array
297             var linksSequence = new ulong[sequence.Length];
298             for (var i = 0; i < sequence.Length; i++)
299             {
300                 linksSequence[i] = FromCharToLink(sequence[i]);
301             }
302             return linksSequence;
303         }
304 
305         /// <summary>
306         /// <para>
307         /// Creates the string to link array groups using the specified sequence.
308         /// </para>
309         /// <para></para>
310         /// </summary>
311         /// <param name="sequence">
312         /// <para>The sequence.</para>
313         /// <para></para>
314         /// </param>
315         /// <returns>
316         /// <para>The result.</para>
317         /// <para></para>
318         /// </returns>
319         [MethodImpl(MethodImplOptions.AggressiveInlining)]
320         public static List<ulong[]> FromStringToLinkArrayGroups(string sequence)
321         {
322             var result = new List<ulong[]>();
323             var offset = 0;
324             while (offset < sequence.Length)
325             {
326                 var currentCategory = CharUnicodeInfo.GetUnicodeCategory(sequence[offset]);
327                 var relativeLength = 1;
328                 var absoluteLength = offset + relativeLength;
329                 while (absoluteLength < sequence.Length &&
330                        currentCategory ==
                           ↪ CharUnicodeInfo.GetUnicodeCategory(sequence[absoluteLength]))
331                 {
332                     relativeLength++;
333                     absoluteLength++;
334                 }
335                 // char array to ulong array
336                 var innerSequence = new ulong[relativeLength];
337                 var maxLength = offset + relativeLength;
338                 for (var i = offset; i < maxLength; i++)
339                 {
340                     innerSequence[i - offset] = FromCharToLink(sequence[i]);
341                 }
342                 result.Add(innerSequence);
343                 offset += relativeLength;
344             }
345             return result;
346         }
347 
348         /// <summary>
349         /// <para>
350         /// Creates the link array to link array groups using the specified array.
351         /// </para>
352         /// <para></para>
353         /// </summary>
354         /// <param name="array">
355         /// <para>The array.</para>
356         /// <para></para>
357         /// </param>
358         /// <returns>
359         /// <para>The result.</para>
360         /// <para></para>
361         /// </returns>
362         [MethodImpl(MethodImplOptions.AggressiveInlining)]
363         public static List<ulong[]> FromLinkArrayToLinkArrayGroups(ulong[] array)
364         {
365             var result = new List<ulong[]>();
366             var offset = 0;
367             while (offset < array.Length)
368             {
369                 var relativeLength = 1;
370                 if (array[offset] <= LastCharLink)
371                 {
372                     var currentCategory =
                        ↪ CharUnicodeInfo.GetUnicodeCategory(FromLinkToChar(array[offset]));
```

```
                    var absoluteLength = offset + relativeLength;
                    while (absoluteLength < array.Length &&
                           array[absoluteLength] <= LastCharLink &&
                           currentCategory == CharUnicodeInfo.GetUnicodeCategory(FromLinkToChar(
                               array[absoluteLength])))
                    {
                        relativeLength++;
                        absoluteLength++;
                    }
                }
                else
                {
                    var absoluteLength = offset + relativeLength;
                    while (absoluteLength < array.Length && array[absoluteLength] > LastCharLink)
                    {
                        relativeLength++;
                        absoluteLength++;
                    }
                }
                // copy array
                var innerSequence = new ulong[relativeLength];
                var maxLength = offset + relativeLength;
                for (var i = offset; i < maxLength; i++)
                {
                    innerSequence[i - offset] = array[i];
                }
                result.Add(innerSequence);
                offset += relativeLength;
            }
            return result;
        }
    }
}
```

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

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

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

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

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="UnicodeSequenceToStringConverter"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="links">
        /// <para>A links.</para>
        /// <para></para>
        /// </param>
        /// <param name="unicodeSequenceCriterionMatcher">
        /// <para>A unicode sequence criterion matcher.</para>
        /// <para></para>
        /// </param>
        /// <param name="sequenceWalker">
        /// <para>A sequence walker.</para>
        /// <para></para>
        /// </param>
        /// <param name="unicodeSymbolToCharConverter">
```

```
45          /// <para>A unicode symbol to char converter.</para>
46          /// <para></para>
47          /// </param>
48          [MethodImpl(MethodImplOptions.AggressiveInlining)]
49          public UnicodeSequenceToStringConverter(ILinks<TLink> links, ICriterionMatcher<TLink>
            ↪  unicodeSequenceCriterionMatcher, ISequenceWalker<TLink> sequenceWalker,
            ↪  IConverter<TLink, char> unicodeSymbolToCharConverter) : base(links)
50          {
51              _unicodeSequenceCriterionMatcher = unicodeSequenceCriterionMatcher;
52              _sequenceWalker = sequenceWalker;
53              _unicodeSymbolToCharConverter = unicodeSymbolToCharConverter;
54          }
55
56          /// <summary>
57          /// <para>
58          /// Converts the source.
59          /// </para>
60          /// <para></para>
61          /// </summary>
62          /// <param name="source">
63          /// <para>The source.</para>
64          /// <para></para>
65          /// </param>
66          /// <exception cref="ArgumentOutOfRangeException">
67          /// <para>Specified link is not a unicode sequence.</para>
68          /// <para></para>
69          /// </exception>
70          /// <returns>
71          /// <para>The string</para>
72          /// <para></para>
73          /// </returns>
74          [MethodImpl(MethodImplOptions.AggressiveInlining)]
75          public string Convert(TLink source)
76          {
77              if (!_unicodeSequenceCriterionMatcher.IsMatched(source))
78              {
79                  throw new ArgumentOutOfRangeException(nameof(source), source, "Specified link is
                    ↪  not a unicode sequence.");
80              }
81              var sequence = _links.GetSource(source);
82              var sb = new StringBuilder();
83              foreach(var character in _sequenceWalker.Walk(sequence))
84              {
85                  sb.Append(_unicodeSymbolToCharConverter.Convert(character));
86              }
87              return sb.ToString();
88          }
89      }
90  }
```

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

```
1   using System;
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.Unicode
9   {
10      /// <summary>
11      /// <para>
12      /// Represents the unicode symbol to char converter.
13      /// </para>
14      /// <para></para>
15      /// </summary>
16      /// <seealso cref="LinksOperatorBase{TLink}"/>
17      /// <seealso cref="IConverter{TLink, char}"/>
18      public class UnicodeSymbolToCharConverter<TLink> : LinksOperatorBase<TLink>,
        ↪  IConverter<TLink, char>
19      {
20          private static readonly UncheckedConverter<TLink, char> _addressToCharConverter =
            ↪  UncheckedConverter<TLink, char>.Default;
21          private readonly IConverter<TLink> _numberToAddressConverter;
22          private readonly ICriterionMatcher<TLink> _unicodeSymbolCriterionMatcher;
23
24          /// <summary>
25          /// <para>
26          /// Initializes a new <see cref="UnicodeSymbolToCharConverter"/> 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="numberToAddressConverter">
35          /// <para>A number to address converter.</para>
36          /// <para></para>
37          /// </param>
38          /// <param name="unicodeSymbolCriterionMatcher">
39          /// <para>A unicode symbol criterion matcher.</para>
40          /// <para></para>
41          /// </param>
42          [MethodImpl(MethodImplOptions.AggressiveInlining)]
43          public UnicodeSymbolToCharConverter(ILinks<TLink> links, IConverter<TLink>
            ↪ numberToAddressConverter, ICriterionMatcher<TLink> unicodeSymbolCriterionMatcher) :
            ↪ base(links)
44          {
45              _numberToAddressConverter = numberToAddressConverter;
46              _unicodeSymbolCriterionMatcher = unicodeSymbolCriterionMatcher;
47          }
48
49          /// <summary>
50          /// <para>
51          /// Converts the source.
52          /// </para>
53          /// <para></para>
54          /// </summary>
55          /// <param name="source">
56          /// <para>The source.</para>
57          /// <para></para>
58          /// </param>
59          /// <exception cref="ArgumentOutOfRangeException">
60          /// <para>Specified link is not a unicode symbol.</para>
61          /// <para></para>
62          /// </exception>
63          /// <returns>
64          /// <para>The char</para>
65          /// <para></para>
66          /// </returns>
67          [MethodImpl(MethodImplOptions.AggressiveInlining)]
68          public char Convert(TLink source)
69          {
70              if (!_unicodeSymbolCriterionMatcher.IsMatched(source))
71              {
72                  throw new ArgumentOutOfRangeException(nameof(source), source, "Specified link is
                    ↪ not a unicode symbol.");
73              }
74              return _addressToCharConverter.Convert(_numberToAddressConverter.Convert(_links.GetS
                ↪ ource(source)));
75          }
76      }
77  }
```

1.55 ./csharp/Platform.Data.Doublets.Sequences/Unicode/UnicodeSymbolsListToUnicodeSequenceConverter.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 unicode symbols list to unicode sequence converter.
13      /// </para>
14      /// <para></para>
15      /// </summary>
16      /// <seealso cref="LinksOperatorBase{TLink}"/>
17      /// <seealso cref="IConverter{IList{TLink}, TLink}"/>
18      public class UnicodeSymbolsListToUnicodeSequenceConverter<TLink> : LinksOperatorBase<TLink>,
        ↪ IConverter<IList<TLink>, TLink>
19      {
20          private readonly ISequenceIndex<TLink> _index;
```

```
21          private readonly IConverter<IList<TLink>, TLink> _listToSequenceLinkConverter;
22          private readonly TLink _unicodeSequenceMarker;
23
24          /// <summary>
25          /// <para>
26          /// Initializes a new <see cref="UnicodeSymbolsListToUnicodeSequenceConverter"/>
            ↪  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="index">
35          /// <para>A index.</para>
36          /// <para></para>
37          /// </param>
38          /// <param name="listToSequenceLinkConverter">
39          /// <para>A list to sequence link converter.</para>
40          /// <para></para>
41          /// </param>
42          /// <param name="unicodeSequenceMarker">
43          /// <para>A unicode sequence marker.</para>
44          /// <para></para>
45          /// </param>
46          [MethodImpl(MethodImplOptions.AggressiveInlining)]
47          public UnicodeSymbolsListToUnicodeSequenceConverter(ILinks<TLink> links,
            ↪  ISequenceIndex<TLink> index, IConverter<IList<TLink>, TLink>
            ↪  listToSequenceLinkConverter, TLink unicodeSequenceMarker) : base(links)
48          {
49              _index = index;
50              _listToSequenceLinkConverter = listToSequenceLinkConverter;
51              _unicodeSequenceMarker = unicodeSequenceMarker;
52          }
53
54          /// <summary>
55          /// <para>
56          /// Initializes a new <see cref="UnicodeSymbolsListToUnicodeSequenceConverter"/>
            ↪  instance.
57          /// </para>
58          /// <para></para>
59          /// </summary>
60          /// <param name="links">
61          /// <para>A links.</para>
62          /// <para></para>
63          /// </param>
64          /// <param name="listToSequenceLinkConverter">
65          /// <para>A list to sequence link converter.</para>
66          /// <para></para>
67          /// </param>
68          /// <param name="unicodeSequenceMarker">
69          /// <para>A unicode sequence marker.</para>
70          /// <para></para>
71          /// </param>
72          [MethodImpl(MethodImplOptions.AggressiveInlining)]
73          public UnicodeSymbolsListToUnicodeSequenceConverter(ILinks<TLink> links,
            ↪  IConverter<IList<TLink>, TLink> listToSequenceLinkConverter, TLink
            ↪  unicodeSequenceMarker)
74              : this(links, new Unindex<TLink>(), listToSequenceLinkConverter,
                ↪  unicodeSequenceMarker) { }
75
76          /// <summary>
77          /// <para>
78          /// Converts the list.
79          /// </para>
80          /// <para></para>
81          /// </summary>
82          /// <param name="list">
83          /// <para>The list.</para>
84          /// <para></para>
85          /// </param>
86          /// <returns>
87          /// <para>The link</para>
88          /// <para></para>
89          /// </returns>
90          [MethodImpl(MethodImplOptions.AggressiveInlining)]
91          public TLink Convert(IList<TLink> list)
```

```
        {
            _index.Add(list);
            var sequence = _listToSequenceLinkConverter.Convert(list);
            return _links.GetOrCreate(sequence, _unicodeSequenceMarker);
        }
    }
}
```

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

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

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

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

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

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

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

namespace Platform.Data.Doublets.Sequences.Walkers
{
    /// <summary>
    /// <para>
    /// Represents the left sequence walker.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="SequenceWalkerBase{TLink}"/>
    public class LeftSequenceWalker<TLink> : SequenceWalkerBase<TLink>
    {
        /// <summary>
        /// <para>
        /// Initializes a new <see cref="LeftSequenceWalker"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="links">
        /// <para>A links.</para>
        /// <para></para>
        /// </param>
        /// <param name="stack">
        /// <para>A stack.</para>
        /// <para></para>
        /// </param>
        /// <param name="isElement">
        /// <para>A is element.</para>
```

```
35          /// <para></para>
36          /// </param>
37          [MethodImpl(MethodImplOptions.AggressiveInlining)]
38          public LeftSequenceWalker(ILinks<TLink> links, IStack<TLink> stack, Func<TLink, bool>
            ↪  isElement) : base(links, stack, isElement) { }
39
40          /// <summary>
41          /// <para>
42          /// Initializes a new <see cref="LeftSequenceWalker"/> instance.
43          /// </para>
44          /// <para></para>
45          /// </summary>
46          /// <param name="links">
47          /// <para>A links.</para>
48          /// <para></para>
49          /// </param>
50          /// <param name="stack">
51          /// <para>A stack.</para>
52          /// <para></para>
53          /// </param>
54          [MethodImpl(MethodImplOptions.AggressiveInlining)]
55          public LeftSequenceWalker(ILinks<TLink> links, IStack<TLink> stack) : base(links, stack,
            ↪  links.IsPartialPoint) { }
56
57          /// <summary>
58          /// <para>
59          /// Gets the next element after pop using the specified element.
60          /// </para>
61          /// <para></para>
62          /// </summary>
63          /// <param name="element">
64          /// <para>The element.</para>
65          /// <para></para>
66          /// </param>
67          /// <returns>
68          /// <para>The link</para>
69          /// <para></para>
70          /// </returns>
71          [MethodImpl(MethodImplOptions.AggressiveInlining)]
72          protected override TLink GetNextElementAfterPop(TLink element) =>
            ↪  _links.GetSource(element);
73
74          /// <summary>
75          /// <para>
76          /// Gets the next element after push using the specified element.
77          /// </para>
78          /// <para></para>
79          /// </summary>
80          /// <param name="element">
81          /// <para>The element.</para>
82          /// <para></para>
83          /// </param>
84          /// <returns>
85          /// <para>The link</para>
86          /// <para></para>
87          /// </returns>
88          [MethodImpl(MethodImplOptions.AggressiveInlining)]
89          protected override TLink GetNextElementAfterPush(TLink element) =>
            ↪  _links.GetTarget(element);
90
91          /// <summary>
92          /// <para>
93          /// Walks the contents using the specified element.
94          /// </para>
95          /// <para></para>
96          /// </summary>
97          /// <param name="element">
98          /// <para>The element.</para>
99          /// <para></para>
100         /// </param>
101         /// <returns>
102         /// <para>An enumerable of t link</para>
103         /// <para></para>
104         /// </returns>
105         [MethodImpl(MethodImplOptions.AggressiveInlining)]
106         protected override IEnumerable<TLink> WalkContents(TLink element)
107         {
108             var links = _links;
```

```
            var parts = links.GetLink(element);
            var start = links.Constants.SourcePart;
            for (var i = parts.Count - 1; i >= start; i--)
            {
                var part = parts[i];
                if (IsElement(part))
                {
                    yield return part;
                }
            }
        }
    }
}
```

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

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

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

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

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

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

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

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

```
        /// </summary>
        /// <param name="sequence">
        /// <para>The sequence.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>An enumerable of t link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public IEnumerable<TLink> Walk(TLink sequence) => ToArray(sequence);

        /// <summary>
        /// <para>
        /// Returns the array using the specified sequence.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="sequence">
        /// <para>The sequence.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link array</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TLink[] ToArray(TLink sequence)
        {
            var length = 1;
            var array = new TLink[length];
            array[0] = sequence;
            if (_isElement(sequence))
            {
                return array;
            }
            bool hasElements;
            do
            {
                length *= 2;
#if USEARRAYPOOL
                var nextArray = ArrayPool.Allocate<ulong>(length);
#else
                var nextArray = new TLink[length];
#endif
                hasElements = false;
                for (var i = 0; i < array.Length; i++)
                {
                    var candidate = array[i];
                    if (_equalityComparer.Equals(array[i], default))
                    {
                        continue;
                    }
                    var doubletOffset = i * 2;
                    if (_isElement(candidate))
                    {
                        nextArray[doubletOffset] = candidate;
                    }
                    else
                    {
                        var links = _links;
                        var link = links.GetLink(candidate);
                        var linkSource = links.GetSource(link);
                        var linkTarget = links.GetTarget(link);
                        nextArray[doubletOffset] = linkSource;
                        nextArray[doubletOffset + 1] = linkTarget;
                        if (!hasElements)
                        {
                            hasElements = !(_isElement(linkSource) && _isElement(linkTarget));
                        }
                    }
                }
#if USEARRAYPOOL
                if (array.Length > 1)
                {
                    ArrayPool.Free(array);
                }
#endif
                array = nextArray;
```

```
            }
            while (hasElements);
            var filledElementsCount = CountFilledElements(array);
            if (filledElementsCount == array.Length)
            {
                return array;
            }
            else
            {
                return CopyFilledElements(array, filledElementsCount);
            }
        }
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static TLink[] CopyFilledElements(TLink[] array, int filledElementsCount)
        {
            var finalArray = new TLink[filledElementsCount];
            for (int i = 0, j = 0; i < array.Length; i++)
            {
                if (!_equalityComparer.Equals(array[i], default))
                {
                    finalArray[j] = array[i];
                    j++;
                }
            }
#if USEARRAYPOOL
                ArrayPool.Free(array);
#endif
            return finalArray;
        }
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static int CountFilledElements(TLink[] array)
        {
            var count = 0;
            for (var i = 0; i < array.Length; i++)
            {
                if (!_equalityComparer.Equals(array[i], default))
                {
                    count++;
                }
            }
            return count;
        }
    }
}
```

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

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

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

namespace Platform.Data.Doublets.Sequences.Walkers
{
    /// <summary>
    /// <para>
    /// Represents the right sequence walker.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="SequenceWalkerBase{TLink}"/>
    public class RightSequenceWalker<TLink> : SequenceWalkerBase<TLink>
    {
        /// <summary>
        /// <para>
        /// Initializes a new <see cref="RightSequenceWalker"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="links">
        /// <para>A links.</para>
        /// <para></para>
        /// </param>
        /// <param name="stack">
        /// <para>A stack.</para>
        /// <para></para>
        /// </param>
        /// <param name="isElement">
```

```
34          /// <para>A is element.</para>
35          /// <para></para>
36          /// </param>
37          [MethodImpl(MethodImplOptions.AggressiveInlining)]
38          public RightSequenceWalker(ILinks<TLink> links, IStack<TLink> stack, Func<TLink, bool>
            ↪  isElement) : base(links, stack, isElement) { }
39
40          /// <summary>
41          /// <para>
42          /// Initializes a new <see cref="RightSequenceWalker"/> instance.
43          /// </para>
44          /// <para></para>
45          /// </summary>
46          /// <param name="links">
47          /// <para>A links.</para>
48          /// <para></para>
49          /// </param>
50          /// <param name="stack">
51          /// <para>A stack.</para>
52          /// <para></para>
53          /// </param>
54          [MethodImpl(MethodImplOptions.AggressiveInlining)]
55          public RightSequenceWalker(ILinks<TLink> links, IStack<TLink> stack) : base(links,
            ↪  stack, links.IsPartialPoint) { }
56
57          /// <summary>
58          /// <para>
59          /// Gets the next element after pop using the specified element.
60          /// </para>
61          /// <para></para>
62          /// </summary>
63          /// <param name="element">
64          /// <para>The element.</para>
65          /// <para></para>
66          /// </param>
67          /// <returns>
68          /// <para>The link</para>
69          /// <para></para>
70          /// </returns>
71          [MethodImpl(MethodImplOptions.AggressiveInlining)]
72          protected override TLink GetNextElementAfterPop(TLink element) =>
            ↪  _links.GetTarget(element);
73
74          /// <summary>
75          /// <para>
76          /// Gets the next element after push using the specified element.
77          /// </para>
78          /// <para></para>
79          /// </summary>
80          /// <param name="element">
81          /// <para>The element.</para>
82          /// <para></para>
83          /// </param>
84          /// <returns>
85          /// <para>The link</para>
86          /// <para></para>
87          /// </returns>
88          [MethodImpl(MethodImplOptions.AggressiveInlining)]
89          protected override TLink GetNextElementAfterPush(TLink element) =>
            ↪  _links.GetSource(element);
90
91          /// <summary>
92          /// <para>
93          /// Walks the contents using the specified element.
94          /// </para>
95          /// <para></para>
96          /// </summary>
97          /// <param name="element">
98          /// <para>The element.</para>
99          /// <para></para>
100         /// </param>
101         /// <returns>
102         /// <para>An enumerable of t link</para>
103         /// <para></para>
104         /// </returns>
105         [MethodImpl(MethodImplOptions.AggressiveInlining)]
106         protected override IEnumerable<TLink> WalkContents(TLink element)
107         {
```

```
            var parts = _links.GetLink(element);
            for (var i = _links.Constants.SourcePart; i < parts.Count; i++)
            {
                var part = parts[i];
                if (IsElement(part))
                {
                    yield return part;
                }
            }
        }
    }
}
```

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

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

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

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

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

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="SequenceWalkerBase"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="links">
        /// <para>A links.</para>
        /// <para></para>
        /// </param>
        /// <param name="stack">
        /// <para>A stack.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
        protected SequenceWalkerBase(ILinks<TLink> links, IStack<TLink> stack) : this(links,
        ↪  stack, links.IsPartialPoint) { }

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

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

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

```
        /// </param>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected abstract TLink GetNextElementAfterPop(TLink element);

        /// <summary>
        /// <para>
        /// Gets the next element after push using the specified element.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="element">
        /// <para>The element.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected abstract TLink GetNextElementAfterPush(TLink 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 abstract IEnumerable<TLink> WalkContents(TLink element);
    }
}
```

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

```
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 TLink = System.UInt64;

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

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

        [Fact]
        public void DecimalMaxValueTest()
        {
            var links = CreateLinks();
            BigInteger bigInteger = new(decimal.MaxValue);
            TLink negativeNumberMarker = links.Create();
            AddressToRawNumberConverter<TLink> addressToRawNumberConverter = new();
```

```
            RawNumberToAddressConverter<TLink> numberToAddressConverter = new();
            BalancedVariantConverter<TLink> listToSequenceConverter = new(links);
            BigIntegerToRawNumberSequenceConverter<TLink> bigIntegerToRawNumberSequenceConverter
            ↪  = new(links, addressToRawNumberConverter, listToSequenceConverter,
            ↪  negativeNumberMarker);
            RawNumberSequenceToBigIntegerConverter<TLink> rawNumberSequenceToBigIntegerConverter
            ↪  = new(links, numberToAddressConverter, negativeNumberMarker);
            var bigIntSequence = bigIntegerToRawNumberSequenceConverter.Convert(bigInteger);
            var bigIntFromSequence =
            ↪  rawNumberSequenceToBigIntegerConverter.Convert(bigIntSequence);
            Assert.Equal(bigInteger, bigIntFromSequence);
        }

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

        [Fact]
        public void ZeroValueTest()
        {
            var links = CreateLinks();
            BigInteger bigInteger = new(0);
            TLink negativeNumberMarker = links.Create();
            AddressToRawNumberConverter<TLink> addressToRawNumberConverter = new();
            RawNumberToAddressConverter<TLink> numberToAddressConverter = new();
            BalancedVariantConverter<TLink> listToSequenceConverter = new(links);
            BigIntegerToRawNumberSequenceConverter<TLink> bigIntegerToRawNumberSequenceConverter
            ↪  = new(links, addressToRawNumberConverter, listToSequenceConverter,
            ↪  negativeNumberMarker);
            RawNumberSequenceToBigIntegerConverter<TLink> rawNumberSequenceToBigIntegerConverter
            ↪  = new(links, numberToAddressConverter, negativeNumberMarker);
            var bigIntSequence = bigIntegerToRawNumberSequenceConverter.Convert(bigInteger);
            var bigIntFromSequence =
            ↪  rawNumberSequenceToBigIntegerConverter.Convert(bigIntSequence);
            Assert.Equal(bigInteger, bigIntFromSequence);
        }

        [Fact]
        public void OneValueTest()
        {
            var links = CreateLinks();
            BigInteger bigInteger = new(1);
            TLink negativeNumberMarker = links.Create();
            AddressToRawNumberConverter<TLink> addressToRawNumberConverter = new();
            RawNumberToAddressConverter<TLink> numberToAddressConverter = new();
            BalancedVariantConverter<TLink> listToSequenceConverter = new(links);
            BigIntegerToRawNumberSequenceConverter<TLink> bigIntegerToRawNumberSequenceConverter
            ↪  = new(links, addressToRawNumberConverter, listToSequenceConverter,
            ↪  negativeNumberMarker);
            RawNumberSequenceToBigIntegerConverter<TLink> rawNumberSequenceToBigIntegerConverter
            ↪  = new(links, numberToAddressConverter, negativeNumberMarker);
            var bigIntSequence = bigIntegerToRawNumberSequenceConverter.Convert(bigInteger);
            var bigIntFromSequence =
            ↪  rawNumberSequenceToBigIntegerConverter.Convert(bigIntSequence);
            Assert.Equal(bigInteger, bigIntFromSequence);
        }
    }
}
```

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

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

```
using Platform.Data.Doublets.Memory;
using Platform.Data.Doublets.Memory.United.Generic;
using Platform.Data.Doublets.Sequences;
using Platform.Data.Doublets.Sequences.HeightProviders;
using Platform.Data.Numbers.Raw;
using Platform.Interfaces;
using Platform.Memory;
using Platform.Numbers;
using Xunit;
using Xunit.Abstractions;
using TLink = System.UInt64;

namespace Platform.Data.Doublets.Sequences.Tests
{
    public class DefaultSequenceAppenderTests
    {
        private readonly ITestOutputHelper _output;

        public DefaultSequenceAppenderTests(ITestOutputHelper output)
        {
            _output = output;
        }
        public static ILinks<TLink> CreateLinks() => CreateLinks<TLink>(new IO.TemporaryFile());

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

        public class ValueCriterionMatcher<TLink> : ICriterionMatcher<TLink>
        {
            public readonly ILinks<TLink> Links;
            public readonly TLink Marker;
            public ValueCriterionMatcher(ILinks<TLink> links, TLink marker)
            {
                Links = links;
                Marker = marker;
            }

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

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

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

```
using Xunit;

namespace Platform.Data.Doublets.Sequences.Tests
{
    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);
            }
        }
    }
}
```

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

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

namespace Platform.Data.Doublets.Sequences.Tests
{
    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);
                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 sequences = new Sequences(links, new SequencesOptions<ulong>() { Walker =
                ↪  new LeveledSequenceWalker<ulong>(links) });

                ExecuteTest(sequences, sequence, sequenceToItsLocalElementLevelsConverter,
                ↪  index, optimalVariantConverter);
            }
        }

        [Fact]
        public static void DictionaryBasedFrequencyStoredOptimalVariantSequenceTest()
        {
            using (var scope = new TempLinksTestScope(useSequences: false))
            {
                var links = scope.Links;

                links.UseUnicode();

                var sequence = UnicodeMap.FromStringToLinkArray(_sequenceExample);

                var totalSequenceSymbolFrequencyCounter = new
                ↪  TotalSequenceSymbolFrequencyCounter<ulong>(links);

                var linkFrequenciesCache = new LinkFrequenciesCache<ulong>(links,
                ↪  totalSequenceSymbolFrequencyCounter);

                var index = new
                ↪  CachedFrequencyIncrementingSequenceIndex<ulong>(linkFrequenciesCache);
                var linkToItsFrequencyNumberConverter = new FrequenciesCacheBasedLinkToItsFreque
                ↪  ncyNumberConverter<ulong>(linkFrequenciesCache);

                var sequenceToItsLocalElementLevelsConverter = new
                ↪  SequenceToItsLocalElementLevelsConverter<ulong>(links,
                ↪  linkToItsFrequencyNumberConverter);
                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 FrequenciesCacheBasedLinkToItsFreque
                    ↪  ncyNumberConverter<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);
```

```
                var arrays = strings.Select(x => x.Select(y =>
                ↪  addressToNumberConverter.Convert(y)).ToArray()).ToArray();
                for (int i = 0; i < arrays.Length; i++)
                {
                    unicodeSequences.Create(arrays[i].ShiftRight());
                }

                var linksCountAfterCreation = links.Count();

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

                var linksCountAfterCompactification = links.Count();

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

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

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

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

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

        [Fact]
        public void DecimalMinValueTest()
        {
            const decimal @decimal = decimal.MinValue;
            var links = CreateLinks();
            TLink negativeNumberMarker = links.Create();
            AddressToRawNumberConverter<TLink> addressToRawNumberConverter = new();
            RawNumberToAddressConverter<TLink> numberToAddressConverter = new();
            BalancedVariantConverter<TLink> balancedVariantConverter = new(links);
            BigIntegerToRawNumberSequenceConverter<TLink> bigIntegerToRawNumberSequenceConverter
            ↪  = new(links, addressToRawNumberConverter, balancedVariantConverter,
            ↪  negativeNumberMarker);
            RawNumberSequenceToBigIntegerConverter<TLink> rawNumberSequenceToBigIntegerConverter
            ↪  = new(links, numberToAddressConverter, negativeNumberMarker);
            DecimalToRationalConverter<TLink> decimalToRationalConverter = new(links,
            ↪  bigIntegerToRawNumberSequenceConverter);
            RationalToDecimalConverter<TLink> rationalToDecimalConverter = new(links,
            ↪  rawNumberSequenceToBigIntegerConverter);
            var rationalNumber = decimalToRationalConverter.Convert(@decimal);
            var decimalFromRational = rationalToDecimalConverter.Convert(rationalNumber);
           Assert.Equal(@decimal, decimalFromRational);
        }

        [Fact]
        public void DecimalMaxValueTest()
        {
            const decimal @decimal = decimal.MaxValue;
```

```
45              var links = CreateLinks();
46              TLink negativeNumberMarker = links.Create();
47              AddressToRawNumberConverter<TLink> addressToRawNumberConverter = new();
48              RawNumberToAddressConverter<TLink> numberToAddressConverter = new();
49              BalancedVariantConverter<TLink> balancedVariantConverter = new(links);
50              BigIntegerToRawNumberSequenceConverter<TLink> bigIntegerToRawNumberSequenceConverter
                ↪  = new(links, addressToRawNumberConverter, balancedVariantConverter,
                ↪  negativeNumberMarker);
51              RawNumberSequenceToBigIntegerConverter<TLink> rawNumberSequenceToBigIntegerConverter
                ↪  = new(links, numberToAddressConverter, negativeNumberMarker);
52              DecimalToRationalConverter<TLink> decimalToRationalConverter = new(links,
                ↪  bigIntegerToRawNumberSequenceConverter);
53              RationalToDecimalConverter<TLink> 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()
61          {
62              const decimal @decimal = 0.5M;
63              var links = CreateLinks();
64              TLink negativeNumberMarker = links.Create();
65              AddressToRawNumberConverter<TLink> addressToRawNumberConverter = new();
66              RawNumberToAddressConverter<TLink> numberToAddressConverter = new();
67              BalancedVariantConverter<TLink> balancedVariantConverter = new(links);
68              BigIntegerToRawNumberSequenceConverter<TLink> bigIntegerToRawNumberSequenceConverter
                ↪  = new(links, addressToRawNumberConverter, balancedVariantConverter,
                ↪  negativeNumberMarker);
69              RawNumberSequenceToBigIntegerConverter<TLink> rawNumberSequenceToBigIntegerConverter
                ↪  = new(links, numberToAddressConverter, negativeNumberMarker);
70              DecimalToRationalConverter<TLink> decimalToRationalConverter = new(links,
                ↪  bigIntegerToRawNumberSequenceConverter);
71              RationalToDecimalConverter<TLink> 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              TLink negativeNumberMarker = links.Create();
83              AddressToRawNumberConverter<TLink> addressToRawNumberConverter = new();
84              RawNumberToAddressConverter<TLink> numberToAddressConverter = new();
85              BalancedVariantConverter<TLink> balancedVariantConverter = new(links);
86              BigIntegerToRawNumberSequenceConverter<TLink> bigIntegerToRawNumberSequenceConverter
                ↪  = new(links, addressToRawNumberConverter, balancedVariantConverter,
                ↪  negativeNumberMarker);
87              RawNumberSequenceToBigIntegerConverter<TLink> rawNumberSequenceToBigIntegerConverter
                ↪  = new(links, numberToAddressConverter, negativeNumberMarker);
88              DecimalToRationalConverter<TLink> decimalToRationalConverter = new(links,
                ↪  bigIntegerToRawNumberSequenceConverter);
89              RationalToDecimalConverter<TLink> 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             TLink negativeNumberMarker = links.Create();
101             AddressToRawNumberConverter<TLink> addressToRawNumberConverter = new();
102             RawNumberToAddressConverter<TLink> numberToAddressConverter = new();
103             BalancedVariantConverter<TLink> balancedVariantConverter = new(links);
104             BigIntegerToRawNumberSequenceConverter<TLink> bigIntegerToRawNumberSequenceConverter
                ↪  = new(links, addressToRawNumberConverter, balancedVariantConverter,
                ↪  negativeNumberMarker);
```

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

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

```
1  using System;
2  using System.Collections.Generic;
3  using System.Diagnostics;
4  using System.Linq;
5  using Xunit;
6  using Platform.Data.Sequences;
7  using Platform.Data.Doublets.Sequences.Converters;
8  using Platform.Data.Doublets.Sequences.Walkers;
9  using Platform.Data.Doublets.Sequences;
10
11 namespace Platform.Data.Doublets.Sequences.Tests
12 {
13     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,
```

```
                                        links.GetSource,
                                        links.GetTarget,
                                        links.IsPartialPoint,
                                        readSequence2.Add);
                sw3.Stop();

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

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

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

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

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

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

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

namespace Platform.Data.Doublets.Sequences.Tests
{
    public static class SequencesTests
    {
        private static readonly LinksConstants<ulong> _constants =
        ↪  Default<LinksConstants<ulong>>.Instance;

        static SequencesTests()
        {
            // Trigger static constructor to not mess with perfomance measurements
            _ = BitString.GetBitMaskFromIndex(1);
        }

        [Fact]
        public static void CreateAllVariantsTest()
        {
            const long sequenceLength = 8;

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

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

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

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

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

                for (var i = 0; i < sequenceLength; i++)
                {
```

```
                links.Delete(sequence[i]);
            }

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

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

    //    const long sequenceLength = 8;

    //    const ulong itself = LinksConstants.Itself;

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

    //        SequencesOptions o = new SequencesOptions();

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

    //        var sequences = new Sequences(links);

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

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

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

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

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

    [Fact]
    public static void AllVariantsSearchTest()
    {
        const long sequenceLength = 8;

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

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

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

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

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

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

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

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

```

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

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

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

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

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

    [Fact]
    public static void BalancedVariantSearchTest()
    {
        const long sequenceLength = 200;

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

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

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

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

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

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

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

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

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

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

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

    [Fact]
    public static void AllPartialVariantsSearchTest()
    {
        const long sequenceLength = 8;

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

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

```
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();
```

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

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

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

                index.Add(sequence);

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

Обозначение пустоты, какое оно? Темнота ли это? Там где отсутствие света, отсутствие фотонов
↪ (носителей света)? Или это то, что полностью отражает свет? Пустой белый лист бумаги? Там
↪ где есть место для нового начала? Разве пустота это не характеристика пространства?
↪ Пространство это то, что можно чем-то наполнить?

[![чёрное пространство, белое
↪ пространство](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/1.png
↪ ""чёрное пространство, белое пространство"")](https://raw.githubusercontent.com/Konard/Links
↪ Platform/master/doc/Intro/1.png)

Что может быть минимальным рисунком, образом, графикой? Может быть это точка? Это ли простейшая
↪ форма? Но есть ли у точки размер? Цвет? Масса? Координаты? Время существования?

[![чёрное пространство, чёрная
↪ точка](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/2.png
↪ ""чёрное пространство, чёрная
↪ точка"")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/2.png)

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

[![белая точка, чёрная
↪ точка](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/3.png ""белая
↪ точка, чёрная
↪ точка"")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/3.png)

А что если мы вообразим движение? Нужно ли время? Каким самым коротким будет путь? Что будет
↪ если этот путь зафиксировать? Запомнить след? Как две точки становятся линией? Чертой?
↪ Гранью? Разделителем? Единицей?

[![две белые точки, чёрная вертикальная
↪ линия](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/4.png ""две
↪ белые точки, чёрная вертикальная
↪ линия"")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/4.png)

Можно ли замкнуть движение? Может ли это быть кругом? Можно ли замкнуть время? Или остаётся
↪ только спираль? Но что если замкнуть предел? Создать ограничение, разделение? Получится
↪ замкнутая область? Полностью отделённая от всего остального? Но что это всё остальное? Что
↪ можно делить? В каком направлении? Ничего или всё? Пустота или полнота? Начало или конец?
↪ Или может быть это единица и ноль? Дуальность? Противоположность? А что будет с кругом если
↪ у него нет размера? Будет ли круг точкой? Точка состоящая из точек?

[![белая вертикальная линия, чёрный
↪ круг](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/5.png ""белая
↪ вертикальная линия, чёрный
↪ круг"")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/5.png)

Как ещё можно использовать грань, черту, линию? А что если она может что-то соединять, может
↪ тогда её нужно повернуть? Почему то, что перпендикулярно вертикальному горизонтально?
↪ Горизонт? Инвертирует ли это смысл? Что такое смысл? Из чего состоит смысл? Существует ли
↪ элементарная единица смысла?

[![белый круг, чёрная горизонтальная
↪ линия](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/6.png ""белый
↪ круг, чёрная горизонтальная
↪ линия"")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/6.png)

Соединять, допустим, а какой смысл в этом есть ещё? Что если помимо смысла ""соединить,
↪ связать"", есть ещё и смысл направления ""от начала к концу""? От предка к потомку? От
↪ родителя к ребёнку? От общего к частному?
```

```
[![белая горизонтальная линия, чёрная горизонтальная стрелка](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/7.png ""белая горизонтальная линия, чёрная горизонтальная стрелка"")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/7.png)

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

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

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

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

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

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

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

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

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

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

...

[![анимация](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/intro-animation-500.gif ""анимация"")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/intro-animation-500.gif)";
        private static readonly string _exampleLoremIpsumText =
            @"Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua.
Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.";

        [Fact]
        public static void CompressionTest()
        {
            using (var scope = new TempLinksTestScope(useSequences: true))
            {
                var links = scope.Links;
                var sequences = scope.Sequences;

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

```
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 FrequenciesCacheBasedLinkToItsFreque
                    ↪  ncyNumberConverter<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
```

```
                }
            }

            var elapsed1 = sw1.Elapsed;

            var balancedVariantConverter = new BalancedVariantConverter<ulong>(scope2.Links);

            var sw2 = Stopwatch.StartNew();

            for (int i = START; i < END; i++)
            {
                var first = balancedVariantConverter.Convert(arrays[i]);
                var second = balancedVariantConverter.Convert(arrays[i]);

                if (first == second)
                {
                    compressed2[i] = first;
                }
            }

            var elapsed2 = sw2.Elapsed;

            Debug.WriteLine($"Compressor: {elapsed1}, Balanced sequence creator:
            ↪  {elapsed2}");

            Assert.True(elapsed1 > elapsed2);

            // Checks
            for (int i = START; i < END; i++)
            {
                var sequence1 = compressed1[i];
                var sequence2 = compressed2[i];

                if (sequence1 != _constants.Null && sequence2 != _constants.Null)
                {
                    var decompress1 = UnicodeMap.FromSequenceLinkToString(sequence1,
                    ↪  scope1.Links);

                    var decompress2 = UnicodeMap.FromSequenceLinkToString(sequence2,
                    ↪  scope2.Links);

                    //var structure1 = scope1.Links.FormatStructure(sequence1, link =>
                    ↪  link.IsPartialPoint());
                    //var structure2 = scope2.Links.FormatStructure(sequence2, link =>
                    ↪  link.IsPartialPoint());

                    //if (sequence1 != Constants.Null && sequence2 != Constants.Null &&
                    ↪  arrays[i].Length > 3)
                    //    Assert.False(structure1 == structure2);

                    Assert.True(strings[i] == decompress1 && decompress1 == decompress2);
                }
            }

            Assert.True((int)(scope1.Links.Count() - UnicodeMap.MapSize) < totalCharacters);
            Assert.True((int)(scope2.Links.Count() - UnicodeMap.MapSize) < totalCharacters);

            Debug.WriteLine($"{(double)(scope1.Links.Count() - UnicodeMap.MapSize) /
            ↪  totalCharacters} | {(double)(scope2.Links.Count() - UnicodeMap.MapSize) /
            ↪  totalCharacters}");

            Assert.True(scope1.Links.Count() <= scope2.Links.Count());

            //compressor1.ValidateFrequencies();
        }
    }

    [Fact]
    public static void RundomNumbersCompressionQualityTest()
    {
        const ulong N = 500;

        //const ulong minNumbers = 10000;
        //const ulong maxNumbers = 20000;

        //var strings = new List<string>();

        //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]);
889                 }
890             }
891         }
892
893         [Fact]
894         public static void AllPossibleConnectionsTest()
895         {
896             const long sequenceLength = 5;
897
898             using (var scope = new TempLinksTestScope(useSequences: true))
899             {
900                 var links = scope.Links;
901                 var sequences = scope.Sequences;
902
903                 var sequence = new ulong[sequenceLength];
904                 for (var i = 0; i < sequenceLength; i++)
905                 {
906                     sequence[i] = links.Create();
907                 }
908
909                 var createResults = sequences.CreateAllVariants2(sequence);
910                 var reverseResults = sequences.CreateAllVariants2(sequence.Reverse().ToArray());
911
912                 for (var i = 0; i < 1; i++)
913                 {
914                     var sw1 = Stopwatch.StartNew();
915                     var searchResults1 = sequences.GetAllConnections(sequence); sw1.Stop();
916
917                     var sw2 = Stopwatch.StartNew();
918                     var searchResults2 = sequences.GetAllConnections1(sequence); sw2.Stop();
919
920                     var sw3 = Stopwatch.StartNew();
921                     var searchResults3 = sequences.GetAllConnections2(sequence); sw3.Stop();
922
923                     var sw4 = Stopwatch.StartNew();
924                     var searchResults4 = sequences.GetAllConnections3(sequence); sw4.Stop();
925
926                     Global.Trash = searchResults3;
927                     Global.Trash = searchResults4; //-V3008
928
929                     var intersection1 = createResults.Intersect(searchResults1).ToList();
930                     Assert.True(intersection1.Count == createResults.Length);
```

```
931
932                     var intersection2 = reverseResults.Intersect(searchResults1).ToList();
933                     Assert.True(intersection2.Count == reverseResults.Length);
934
935                     var intersection0 = searchResults1.Intersect(searchResults2).ToList();
936                     Assert.True(intersection0.Count == searchResults2.Count);
937
938                     var intersection3 = searchResults2.Intersect(searchResults3).ToList();
939                     Assert.True(intersection3.Count == searchResults3.Count);
940
941                     var intersection4 = searchResults3.Intersect(searchResults4).ToList();
942                     Assert.True(intersection4.Count == searchResults4.Count);
943                 }
944
945                 for (var i = 0; i < sequenceLength; i++)
946                 {
947                     links.Delete(sequence[i]);
948                 }
949             }
950         }
951
952         [Fact(Skip = "Correct implementation is pending")]
953         public static void CalculateAllUsagesTest()
954         {
955             const long sequenceLength = 3;
956
957             using (var scope = new TempLinksTestScope(useSequences: true))
958             {
959                 var links = scope.Links;
960                 var sequences = scope.Sequences;
961
962                 var sequence = new ulong[sequenceLength];
963                 for (var i = 0; i < sequenceLength; i++)
964                 {
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 }
```

1.68 ./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  }
```

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

```
1   using System.Collections.Generic;
2   using Xunit;
3   using Platform.Ranges;
4   using Platform.Numbers;
5   using Platform.Random;
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<TLink>(this ILinks<TLink>
            ↪ links, int maximumOperationsPerCycle)
158         {
159             var comparer = Comparer<TLink>.Default;
160             var addressToUInt64Converter = CheckedConverter<TLink, ulong>.Default;
161             var uInt64ToAddressConverter = CheckedConverter<ulong, TLink>.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                         TLink source = uInt64ToAddressConverter.Convert(random.NextUInt64(linksA
                            ↪ ddressRange));
175                         TLink target = uInt64ToAddressConverter.Convert(random.NextUInt64(linksA
                            ↪ ddressRange));
                            ↪ //-V3086
176                         var resultLink = links.GetOrCreate(source, target);
177                         if (comparer.Compare(resultLink,
                            ↪ uInt64ToAddressConverter.Convert(linksCount)) > 0)
178                         {
179                             created++;
180                         }
181                     }
```

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

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

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.IO;
using System.Text;
using System.Threading;
using System.Threading.Tasks;
using Xunit;
using Platform.Disposables;
using Platform.Ranges;
using Platform.Random;
using Platform.Timestamps;
using Platform.Reflection;
using Platform.Singletons;
using Platform.Scopes;
using Platform.Counters;
using Platform.Diagnostics;
using Platform.IO;
using Platform.Memory;
using Platform.Data.Doublets.Decorators;
using Platform.Data.Doublets.Memory.United.Specific;

namespace Platform.Data.Doublets.Sequences.Tests
{
    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>>()).TestMultipleRandomCreationsAndDeleti⌋
                ↪  ons(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);
```

```
56
57                  l2 = links.Update(l2, l1);
58
59                  links.Delete(l2);
60
61                  Global.Trash = links.Count();
62
63                  links.Unsync.DisposeIfPossible(); // Close links to access log
64
65                  Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(scop
                    ↪  e.TempTransactionLogFilename);
66              }
67          }
68
69          [Fact]
70          public static void BasicTransactionLogTest()
71          {
72              using (var scope = new TempLinksTestScope(useLog: true))
73              {
74                  var links = scope.Links;
75                  var l1 = links.Create();
76                  var l2 = links.Create();
77
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>(scop
                    ↪  e.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>(s
                    ↪  cope.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)((LinksDisposableDecor
                        ↪  atorBase<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.Transi
                            ↪  tion>(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>(l
                    ↪  astScope.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.Transition>(
                        ↪  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>(last
                    ↪ Scope.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>(tempTran
                ↪ sactionLogFilename);
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>(tempTran
                ↪ sactionLogFilename);
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>(tempTran
                ↪  sactionLogFilename);
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>(temp
                    ↪  TransactionLogFilename);
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>(temp
                    ↪  TransactionLogFilename);
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();
508
509             ConsoleHelpers.Debug("C S D");
510
511             for (int i = 0; i < loops; i++)
512             {
513                 ConsoleHelpers.Debug("{0} {1} {2}", creationMeasurements[i],
    ↪  searchMeasuremets[i], deletionMeasurements[i]);
514             }
515
516             ConsoleHelpers.Debug("C S D (no overhead)");
517
518             for (int i = 0; i < loops; i++)
519             {
520                 ConsoleHelpers.Debug("{0} {1} {2}", creationMeasurements[i] - stepLoopOverhead,
    ↪  searchMeasuremets[i] - stepLoopOverhead, deletionMeasurements[i] - stepLoopOverhead);
521             }
522
523             ConsoleHelpers.Debug("All tests done. Total links left in database: {0}.",
    ↪  links.Total);
524         }
525
526         private static void CreatePoints(this Platform.Links.Data.Core.Doublets.Links links, long
    ↪  amountToCreate)
527         {
528             for (long i = 0; i < amountToCreate; i++)
529                 links.Create(0, 0);
530         }
531
532          private static TimeSpan GetBaseRandomLoopOverhead(long loops)
533          {
534              return Measure(() =>
535              {
536                  ulong maxValue = RandomHelpers.DefaultFactory.NextUInt64();
537                  ulong result = 0;
538                  for (long i = 0; i < loops; i++)
539                  {
540                      var source = RandomHelpers.DefaultFactory.NextUInt64(maxValue);
541                      var target = RandomHelpers.DefaultFactory.NextUInt64(maxValue);
542
543                      result += maxValue + source + target;
544                  }
545                  Global.Trash = result;
546              });
547          }
548           */
549
550          [Fact(Skip = "performance test")]
551          public static void GetSourceTest()
552          {
553              using (var scope = new TempLinksTestScope())
554              {
555                  var links = scope.Links;
556                  ConsoleHelpers.Debug("Testing GetSource function with {0} Iterations.",
                     ↪  Iterations);
557
558                  ulong counter = 0;
559
560                  //var firstLink = links.First();
561                  // Создаём одну связь, из которой будет производить считывание
562                  var firstLink = links.Create();
563
564                  var sw = Stopwatch.StartNew();
565
566                  // Тестируем саму функцию
567                  for (ulong i = 0; i < Iterations; i++)
568                  {
569                      counter += links.GetSource(firstLink);
570                  }
571
572                  var elapsedTime = sw.Elapsed;
```

```
573
574                 var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;
575
576                 // Удаляем связь, из которой производилось считывание
577                 links.Delete(firstLink);
578
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);
```

```
        }*/

        [Fact(Skip = "useless: 0(0), was dependent on creation tests")]
        public static void TestRandomSearchAll()
        {
            using (var scope = new TempLinksTestScope())
            {
                var links = scope.Links;
                ulong counter = 0;

                var maxLink = links.Count();

                var iterations = links.Count();

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

                var sw = Stopwatch.StartNew();

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

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

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

                var elapsedTime = sw.Elapsed;

                var iterationsPerSecond = iterations / elapsedTime.TotalSeconds;

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

        [Fact(Skip = "useless: 0(0), was dependent on creation tests")]
        public static void TestEach()
        {
            using (var scope = new TempLinksTestScope())
            {
                var links = scope.Links;

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

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

                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.");

                var sw = Stopwatch.StartNew();

```

```
                //foreach (var link in links)
                //{
                //    counter++;
                //}

                var elapsedTime = sw.Elapsed;

                var linksPerSecond = (double)counter / elapsedTime.TotalSeconds;

                ConsoleHelpers.Debug("{0} Iterations of Foreach's handler block done in {1} ({2} links per second)", counter, elapsedTime, (long)linksPerSecond);
            }

            File.Delete(tempFilename);
        }
        */

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

            using (var links = new Platform.Links.Data.Core.Doublets.Links(tempFilename, DefaultLinksSizeStep))
            {

                long counter = 0;

                ConsoleHelpers.Debug("Testing parallel foreach through links.");

                var sw = Stopwatch.StartNew();

                //Parallel.ForEach((IEnumerable<ulong>)links, x =>
                //{
                //    Interlocked.Increment(ref counter);
                //});

                var elapsedTime = sw.Elapsed;

                var linksPerSecond = (double)counter / elapsedTime.TotalSeconds;

                ConsoleHelpers.Debug("{0} Iterations of Parallel Foreach's handler block done in {1} ({2} links per second)", counter, elapsedTime, (long)linksPerSecond);
            }

            File.Delete(tempFilename);
        }
        */

        [Fact(Skip = "performance test")]
        public static void Create64BillionLinks()
        {
            using (var scope = new TempLinksTestScope())
            {
                var links = scope.Links;
                var linksBeforeTest = links.Count();

                long linksToCreate = 64 * 1024 * 1024 / UInt64UnitedMemoryLinks.LinkSizeInBytes;

                ConsoleHelpers.Debug("Creating {0} links.", linksToCreate);

                var elapsedTime = Performance.Measure(() =>
                {
                    for (long i = 0; i < linksToCreate; i++)
                    {
                        links.Create();
                    }
                });

                var linksCreated = links.Count() - linksBeforeTest;
                var linksPerSecond = linksCreated / elapsedTime.TotalSeconds;

                ConsoleHelpers.Debug("Current links count: {0}.", links.Count());

                ConsoleHelpers.Debug("{0} links created in {1} ({2} links per second)",
                    linksCreated, elapsedTime,
                    (long)linksPerSecond);
            }
```

```
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: 0(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 }
```

1.71 ./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 TLink = System.UInt64;
9
10 namespace Platform.Data.Doublets.Sequences.Tests
11 {
12     public class Uint64LinksExtensionsTests
13     {
14         public static ILinks<TLink> CreateLinks() => CreateLinks<TLink>(new
           ↪ Platform.IO.TemporaryFile());
15
16         public static ILinks<TLink> CreateLinks<TLink>(string dataDBFilename)
17         {
18             var linksConstants = new LinksConstants<TLink>(enableExternalReferencesSupport:
               ↪ true);
19             return new UnitedMemoryLinks<TLink>(new
               ↪ FileMappedResizableDirectMemory(dataDBFilename),
               ↪ UnitedMemoryLinks<TLink>.DefaultLinksSizeStep, linksConstants,
               ↪ IndexTreeType.Default);
20         }
```

```
        [Fact]
        public void FormatStructureWithExternalReferenceTest()
        {
            ILinks<TLink> links = CreateLinks();
            TLink zero = default;
            var one = Arithmetic.Increment(zero);
            var markerIndex = one;
            var meaningRoot = links.GetOrCreate(markerIndex, markerIndex);
            var numberMarker = links.GetOrCreate(meaningRoot, Arithmetic.Increment(ref
            ↪  markerIndex));
            AddressToRawNumberConverter<TLink> addressToNumberConverter = new();
            var numberAddress = addressToNumberConverter.Convert(1);
            var numberLink = links.GetOrCreate(numberMarker, numberAddress);
            var linkNotation = links.FormatStructure(numberLink, link => link.IsFullPoint(),
            ↪  true);
            Assert.Equal("(3:(2:1 2) 18446744073709551615)", linkNotation);
        }
    }
}
```

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

```
using Xunit;
using Platform.Random;
using Platform.Data.Doublets.Numbers.Unary;

namespace Platform.Data.Doublets.Sequences.Tests
{
    public static class UnaryNumberConvertersTests
    {
        [Fact]
        public static void ConvertersTest()
        {
            using (var scope = new TempLinksTestScope())
            {
                const int N = 10;
                var links = scope.Links;
                var meaningRoot = links.CreatePoint();
                var one = links.CreateAndUpdate(meaningRoot, links.Constants.Itself);
                var powerOf2ToUnaryNumberConverter = new
                ↪  PowerOf2ToUnaryNumberConverter<ulong>(links, one);
                var toUnaryNumberConverter = new AddressToUnaryNumberConverter<ulong>(links,
                ↪  powerOf2ToUnaryNumberConverter);
                var random = new System.Random(0);
                ulong[] numbers = new ulong[N];
                ulong[] unaryNumbers = new ulong[N];
                for (int i = 0; i < N; i++)
                {
                    numbers[i] = random.NextUInt64();
                    unaryNumbers[i] = toUnaryNumberConverter.Convert(numbers[i]);
                }
                var fromUnaryNumberConverterUsingOrOperation = new
                ↪  UnaryNumberToAddressOrOperationConverter<ulong>(links,
                ↪  powerOf2ToUnaryNumberConverter);
                var fromUnaryNumberConverterUsingAddOperation = new
                ↪  UnaryNumberToAddressAddOperationConverter<ulong>(links, one);
                for (int i = 0; i < N; i++)
                {
                    Assert.Equal(numbers[i],
                    ↪  fromUnaryNumberConverterUsingOrOperation.Convert(unaryNumbers[i]));
                    Assert.Equal(numbers[i],
                    ↪  fromUnaryNumberConverterUsingAddOperation.Convert(unaryNumbers[i]));
                }
            }
        }
    }
}
```

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

```
using Xunit;
using Platform.Converters;
using Platform.Memory;
using Platform.Reflection;
using Platform.Scopes;
using Platform.Data.Numbers.Raw;
using Platform.Data.Doublets.Incrementers;
using Platform.Data.Doublets.Numbers.Unary;
using Platform.Data.Doublets.PropertyOperators;
using Platform.Data.Doublets.Sequences.Converters;
```

```
using Platform.Data.Doublets.Sequences.Indexes;
using Platform.Data.Doublets.Sequences.Walkers;
using Platform.Data.Doublets.Unicode;
using Platform.Data.Doublets.Memory.United.Generic;
using Platform.Data.Doublets.CriterionMatchers;

namespace Platform.Data.Doublets.Sequences.Tests
{
    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);

```

```
76                  var powerOf2ToUnaryNumberConverter = new
                    ↪  PowerOf2ToUnaryNumberConverter<ulong>(links, unaryOne);
77                  var addressToUnaryNumberConverter = new
                    ↪  AddressToUnaryNumberConverter<ulong>(links, powerOf2ToUnaryNumberConverter);
78                  var charToUnicodeSymbolConverter = new
                    ↪  CharToUnicodeSymbolConverter<ulong>(links, addressToUnaryNumberConverter,
                    ↪  unicodeSymbolMarker);
79
80                  var unaryNumberToAddressConverter = new
                    ↪  UnaryNumberToAddressOrOperationConverter<ulong>(links,
                    ↪  powerOf2ToUnaryNumberConverter);
81                  var unaryNumberIncrementer = new UnaryNumberIncrementer<ulong>(links, unaryOne);
82                  var frequencyIncrementer = new FrequencyIncrementer<ulong>(links,
                    ↪  frequencyMarker, unaryOne, unaryNumberIncrementer);
83                  var frequencyPropertyOperator = new PropertyOperator<ulong>(links,
                    ↪  frequencyPropertyMarker, frequencyMarker);
84                  var index = new FrequencyIncrementingSequenceIndex<ulong>(links,
                    ↪  frequencyPropertyOperator, frequencyIncrementer);
85                  var linkToItsFrequencyNumberConverter = new
                    ↪  LinkToItsFrequencyNumberConveter<ulong>(links, frequencyPropertyOperator,
                    ↪  unaryNumberToAddressConverter);
86                  var sequenceToItsLocalElementLevelsConverter = new
                    ↪  SequenceToItsLocalElementLevelsConverter<ulong>(links,
                    ↪  linkToItsFrequencyNumberConverter);
87                  var optimalVariantConverter = new OptimalVariantConverter<ulong>(links,
                    ↪  sequenceToItsLocalElementLevelsConverter);
88
89                  var stringToUnicodeSequenceConverter = new
                    ↪  StringToUnicodeSequenceConverter<ulong>(links, charToUnicodeSymbolConverter,
                    ↪  index, optimalVariantConverter, unicodeSequenceMarker);
90
91                  var originalString = "Hello";
92
93                  var unicodeSequenceLink =
                    ↪  stringToUnicodeSequenceConverter.Convert(originalString);
94
95                  var unicodeSymbolCriterionMatcher = new TargetMatcher<ulong>(links,
                    ↪  unicodeSymbolMarker);
96                  var unicodeSymbolToCharConverter = new
                    ↪  UnicodeSymbolToCharConverter<ulong>(links, unaryNumberToAddressConverter,
                    ↪  unicodeSymbolCriterionMatcher);
97
98                  var unicodeSequenceCriterionMatcher = new TargetMatcher<ulong>(links,
                    ↪  unicodeSequenceMarker);
99
100                 var sequenceWalker = new LeveledSequenceWalker<ulong>(links,
                    ↪  unicodeSymbolCriterionMatcher.IsMatched);
101
102                 var unicodeSequenceToStringConverter = new
                    ↪  UnicodeSequenceToStringConverter<ulong>(links,
                    ↪  unicodeSequenceCriterionMatcher, sequenceWalker,
                    ↪  unicodeSymbolToCharConverter);
103
104                 var resultingString =
                    ↪  unicodeSequenceToStringConverter.Convert(unicodeSequenceLink);
105
106                 Assert.Equal(originalString, resultingString);
107             }
108         }
109     }
110 }
```

Index