1.1 ./csharp/Platform.Data.Triplets/CharacterHelpers.cs

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

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

namespace Platform.Data.Triplets
{
    // TODO: Split logic of Latin and Cyrillic alphabets into different files if possible
    public static class CharacterHelpers
    {
        public enum CharacterMapping : long
        {
            LatinAlphabet = 100,
            CyrillicAlphabet
        }

        private const char FirstLowerCaseLatinLetter = 'a';
        private const char LastLowerCaseLatinLetter = 'z';
        private const char FirstUpperCaseLatinLetter = 'A';
        private const char LastUpperCaseLatinLetter = 'Z';
        private const char FirstLowerCaseCyrillicLetter = 'а';
        private const char LastLowerCaseCyrillicLetter = 'я';
        private const char FirstUpperCaseCyrillicLetter = 'А';
        private const char LastUpperCaseCyrillicLetter = 'Я';
        private const char YoLowerCaseCyrillicLetter = 'ё';
        private const char YoUpperCaseCyrillicLetter = 'Ё';

        private static Link[] _charactersToLinks;
        private static Dictionary<Link, char> _linksToCharacters;

        static CharacterHelpers() => Create();

        private static void Create()
        {
            _charactersToLinks = new Link[char.MaxValue];
            _linksToCharacters = new Dictionary<Link, char>();
            // Create or restore characters
            CreateLatinAlphabet();
            CreateCyrillicAlphabet();
            RegisterExistingCharacters();
        }

        private static void RegisterExistingCharacters() =>
        ↪ Net.Character.WalkThroughReferersAsSource(referer =>
        ↪ RegisterExistingCharacter(referer));

        private static void RegisterExistingCharacter(Link character)
        {
            if (character.Source == Net.Character && character.Linker == Net.ThatHas)
            {
                var code = character.Target;
                if (code.Source == Net.Code && code.Linker == Net.ThatIsRepresentedBy)
                {
                    var charCode = (char)LinkConverter.ToNumber(code.Target);
                    _charactersToLinks[charCode] = character;
                    _linksToCharacters[character] = charCode;
                }
            }
        }

        public static void Recreate() => Create();

        private static void CreateLatinAlphabet()
        {
            var lettersCharacters = new[]
            {
                'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j',
                'k', 'l', 'm', 'n', 'o', 'p', 'q', 'r', 's', 't',
                'u', 'v', 'w', 'x', 'y', 'z'
            };
            CreateAlphabet(lettersCharacters, "latin alphabet", CharacterMapping.LatinAlphabet);
        }

        private static void CreateCyrillicAlphabet()
        {
            var lettersCharacters = new[]
            {
                'а', 'б', 'в', 'г', 'д', 'е', 'ё', 'ж', 'з', 'и',
```

```
            'й', 'к', 'л', 'м', 'н', 'о', 'п', 'р', 'с', 'т',
            'у', 'ф', 'х', 'ц', 'ч', 'ш', 'щ', 'ъ', 'ы', 'ь',
            'э', 'ю', 'я'
        };
        CreateAlphabet(lettersCharacters, "cyrillic alphabet",
        ↪  CharacterMapping.CyrillicAlphabet);
    }

    private static void CreateAlphabet(char[] lettersCharacters, string alphabetName,
    ↪  CharacterMapping mapping)
    {
        if (Link.TryGetMapped(mapping, out Link alphabet))
        {
            var letters = alphabet.Target;
            letters.WalkThroughSequence(letter =>
            {
                var lowerCaseLetter = Link.Search(Net.LowerCase, Net.Of, letter);
                var upperCaseLetter = Link.Search(Net.UpperCase, Net.Of, letter);
                if (lowerCaseLetter != null && upperCaseLetter != null)
                {
                    RegisterExistingLetter(lowerCaseLetter);
                    RegisterExistingLetter(upperCaseLetter);
                }
                else
                {
                    RegisterExistingLetter(letter);
                }
            });
        }
        else
        {
            alphabet = Net.CreateMappedThing(mapping);
            var letterOfAlphabet = Link.Create(Net.Letter, Net.Of, alphabet);
            var lettersLinks = new Link[lettersCharacters.Length];
            GenerateAlphabetBasis(ref alphabet, ref letterOfAlphabet, lettersLinks);
            for (var i = 0; i < lettersCharacters.Length; i++)
            {
                var lowerCaseCharacter = lettersCharacters[i];
                SetLetterCodes(lettersLinks[i], lowerCaseCharacter, out Link lowerCaseLink,
                ↪  out Link upperCaseLink);
                _charactersToLinks[lowerCaseCharacter] = lowerCaseLink;
                _linksToCharacters[lowerCaseLink] = lowerCaseCharacter;
                if (upperCaseLink != null)
                {
                    var upperCaseCharacter = char.ToUpper(lowerCaseCharacter);
                    _charactersToLinks[upperCaseCharacter] = upperCaseLink;
                    _linksToCharacters[upperCaseLink] = upperCaseCharacter;
                }
            }
            alphabet.SetName(alphabetName);
            for (var i = 0; i < lettersCharacters.Length; i++)
            {
                var lowerCaseCharacter = lettersCharacters[i];
                var upperCaseCharacter = char.ToUpper(lowerCaseCharacter);
                if (lowerCaseCharacter != upperCaseCharacter)
                {
                    lettersLinks[i].SetName("{" + upperCaseCharacter + " " +
                    ↪  lowerCaseCharacter + "}");
                }
                else
                {
                    lettersLinks[i].SetName("{" + lowerCaseCharacter + "}");
                }
            }
        }
    }

    private static void RegisterExistingLetter(Link letter)
    {
        letter.WalkThroughRefererersAsSource(referer =>
            {
                if (referer.Linker == Net.Has)
                {
                    var target = referer.Target;
                    if (target.Source == Net.Code && target.Linker ==
                    ↪  Net.ThatIsRepresentedBy)
                    {
                        var charCode = (char)LinkConverter.ToNumber(target.Target);
```

```
                        _charactersToLinks[charCode] = letter;
                        _linksToCharacters[letter] = charCode;
                    }
                }
            });
        }

        private static void GenerateAlphabetBasis(ref Link alphabet, ref Link letterOfAlphabet,
        ↪  Link[] letters)
        {
            // Принцип, на примере латинского алфавита.
            //latin alphabet: alphabet that consists of a and b and c and ... and z.
            //a: letter of latin alphabet that is before b.
            //b: letter of latin alphabet that is between (a and c).
            //c: letter of latin alphabet that is between (b and e).
            //...
            //y: letter of latin alphabet that is between (x and z).
            //z: letter of latin alphabet that is after y.
            const int firstLetterIndex = 0;
            for (var i = firstLetterIndex; i < letters.Length; i++)
            {
                letters[i] = Net.CreateThing();
            }
            var lastLetterIndex = letters.Length - 1;
            Link.Update(ref letters[firstLetterIndex], letterOfAlphabet, Net.ThatIsBefore,
            ↪  letters[firstLetterIndex + 1]);
            Link.Update(ref letters[lastLetterIndex], letterOfAlphabet, Net.ThatIsAfter,
            ↪  letters[lastLetterIndex - 1]);
            const int secondLetterIndex = firstLetterIndex + 1;
            for (var i = secondLetterIndex; i < lastLetterIndex; i++)
            {
                Link.Update(ref letters[i], letterOfAlphabet, Net.ThatIsBetween, letters[i - 1]
                ↪  & letters[i + 1]);
            }
            Link.Update(ref alphabet, Net.Alphabet, Net.ThatConsistsOf,
            ↪  LinkConverter.FromList(letters));
        }

        private static void SetLetterCodes(Link letter, char lowerCaseCharacter, out Link
        ↪  lowerCase, out Link upperCase)
        {
            var upperCaseCharacter = char.ToUpper(lowerCaseCharacter);
            if (upperCaseCharacter != lowerCaseCharacter)
            {
                lowerCase = Link.Create(Net.LowerCase, Net.Of, letter);
                var lowerCaseCharacterCode = Link.Create(Net.Code, Net.ThatIsRepresentedBy,
                ↪  LinkConverter.FromNumber(lowerCaseCharacter));
                Link.Create(lowerCase, Net.Has, lowerCaseCharacterCode);
                upperCase = Link.Create(Net.UpperCase, Net.Of, letter);
                var upperCaseCharacterCode = Link.Create(Net.Code, Net.ThatIsRepresentedBy,
                ↪  LinkConverter.FromNumber(upperCaseCharacter));
                Link.Create(upperCase, Net.Has, upperCaseCharacterCode);
            }
            else
            {
                lowerCase = letter;
                upperCase = null;
                Link.Create(letter, Net.Has, Link.Create(Net.Code, Net.ThatIsRepresentedBy,
                ↪  LinkConverter.FromNumber(lowerCaseCharacter)));
            }
        }

        private static Link CreateSimpleCharacterLink(char character) =>
        ↪  Link.Create(Net.Character, Net.ThatHas, Link.Create(Net.Code,
        ↪  Net.ThatIsRepresentedBy, LinkConverter.FromNumber(character)));

        private static bool IsLetterOfLatinAlphabet(char character)
            => (character >= FirstLowerCaseLatinLetter && character <= LastLowerCaseLatinLetter)
            || (character >= FirstUpperCaseLatinLetter && character <= LastUpperCaseLatinLetter);

        private static bool IsLetterOfCyrillicAlphabet(char character)
            => (character >= FirstLowerCaseCyrillicLetter && character <=
            ↪  LastLowerCaseCyrillicLetter)
            || (character >= FirstUpperCaseCyrillicLetter && character <=
            ↪  LastUpperCaseCyrillicLetter)
            || character == YoLowerCaseCyrillicLetter || character == YoUpperCaseCyrillicLetter;

        public static Link FromChar(char character)
```

```
        {
            if (_charactersToLinks[character] == null)
            {
                if (IsLetterOfLatinAlphabet(character))
                {
                    CreateLatinAlphabet();
                    return _charactersToLinks[character];
                }
                else if (IsLetterOfCyrillicAlphabet(character))
                {
                    CreateCyrillicAlphabet();
                    return _charactersToLinks[character];
                }
                else
                {
                    var simpleCharacter = CreateSimpleCharacterLink(character);
                    _charactersToLinks[character] = simpleCharacter;
                    _linksToCharacters[simpleCharacter] = character;
                    return simpleCharacter;
                }
            }
            else
            {
                return _charactersToLinks[character];
            }
        }

        public static char ToChar(Link link)
        {
            if (!_linksToCharacters.TryGetValue(link, out char @char))
            {
                throw new ArgumentOutOfRangeException(nameof(link), "Указанная связь не
                ↪  является символом.");
            }
            return @char;
        }

        public static bool IsChar(Link link) => link != null &&
        ↪  _linksToCharacters.ContainsKey(link);
    }
}
```

1.2 ./csharp/Platform.Data.Triplets/GexfExporter.cs

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Text;
using System.Xml;
using Platform.Collections.Sets;
using Platform.Communication.Protocol.Gexf;
using GexfNode = Platform.Communication.Protocol.Gexf.Node;

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

namespace Platform.Data.Triplets
{
    public static class GexfExporter
    {
        private const string SourceLabel = "Source";
        private const string LinkerLabel = "Linker";
        private const string TargetLabel = "Target";

        public static string ToXml()
        {
            var sb = new StringBuilder();
            using (var writer = XmlWriter.Create(sb))
            {
                WriteXml(writer, CollectLinks());
            }
            return sb.ToString();
        }

        public static void ToFile(string path)
        {
            using (var file = File.OpenWrite(path))
            using (var writer = XmlWriter.Create(file))
            {
                WriteXml(writer, CollectLinks());
            }
```

```
        }

        public static void ToFile(string path, Func<Link, bool> filter)
        {
            using (var file = File.OpenWrite(path))
            using (var writer = XmlWriter.Create(file))
            {
                WriteXml(writer, CollectLinks(filter));
            }
        }

        private static HashSet<Link> CollectLinks(Func<Link, bool> linkMatch)
        {
            var matchingLinks = new HashSet<Link>();
            Link.WalkThroughAllLinks(link =>
            {
                if (linkMatch(link))
                {
                    matchingLinks.Add(link);
                }
            });
            return matchingLinks;
        }

        private static HashSet<Link> CollectLinks()
        {
            var matchingLinks = new HashSet<Link>();
            Link.WalkThroughAllLinks(matchingLinks.AddAndReturnVoid);
            return matchingLinks;
        }

        private static void WriteXml(XmlWriter writer, HashSet<Link> matchingLinks)
        {
            var edgesCounter = 0;
            Gexf.WriteXml(writer,
            () => // nodes
            {
                foreach (var matchingLink in matchingLinks)
                {
                    GexfNode.WriteXml(writer, matchingLink.ToInt(), matchingLink.ToString());
                }
            },
            () => // edges
            {
                foreach (var matchingLink in matchingLinks)
                {
                    if (matchingLinks.Contains(matchingLink.Source))
                    {
                        Edge.WriteXml(writer, edgesCounter++, matchingLink.ToInt(),
                        ↪  matchingLink.Source.ToInt(), SourceLabel);
                    }
                    if (matchingLinks.Contains(matchingLink.Linker))
                    {
                        Edge.WriteXml(writer, edgesCounter++, matchingLink.ToInt(),
                        ↪  matchingLink.Linker.ToInt(), LinkerLabel);
                    }
                    if (matchingLinks.Contains(matchingLink.Target))
                    {
                        Edge.WriteXml(writer, edgesCounter++, matchingLink.ToInt(),
                        ↪  matchingLink.Target.ToInt(), TargetLabel);
                    }
                }
            });
        }
    }
}
```

1.3 ./csharp/Platform.Data.Triplets/ILink.cs

```
using System;

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

namespace Platform.Data.Triplets
{
    internal partial interface ILink<TLink>
        where TLink : ILink<TLink>
    {
        TLink Source { get; }
```

```
        TLink Linker { get; }
        TLink Target { get; }
    }

    internal partial interface ILink<TLink>
        where TLink : ILink<TLink>
    {
        bool WalkThroughReferersAsLinker(Func<TLink, bool> walker);
        bool WalkThroughReferersAsSource(Func<TLink, bool> walker);
        bool WalkThroughReferersAsTarget(Func<TLink, bool> walker);
        void WalkThroughReferers(Func<TLink, bool> walker);
    }

    internal partial interface ILink<TLink>
        where TLink : ILink<TLink>
    {
        void WalkThroughReferersAsLinker(Action<TLink> walker);
        void WalkThroughReferersAsSource(Action<TLink> walker);
        void WalkThroughReferersAsTarget(Action<TLink> walker);
        void WalkThroughReferers(Action<TLink> walker);
    }
}
/*
using System;
namespace NetLibrary
{
    interface ILink
    {
        // Статические методы (общие для всех связей)
        public static ILink Create(ILink source, ILink linker, ILink target);
        public static void Update(ref ILink link, ILink newSource, ILink newLinker, ILink
↪  newTarget);
        public static void Delete(ref ILink link);
        public static ILink Search(ILink source, ILink linker, ILink target);
    }
}
*/
/*
Набор функций, который необходим для работы с сущностью Link:

(Работа со значением сущности Link, значение состоит из 3-х частей, также сущностей Link)
1. Получить адрес "начальной" сущности Link. (Получить адрес из поля Source)
2. Получить адрес сущности Link, которая играет роль связки между "начальной" и "конечной"
↪  сущностями Link. (Получить адрес из поля Linker)
3. Получить адрес "конечной" сущности Link. (Получить адрес из поля Target)

4. Пройтись по всем сущностями Link, которые ссылаются на сущность Link с указанным адресом, и у
↪  которых поле Source равно этому адресу.
5. Пройтись по всем сущностями Link, которые ссылаются на сущность Link с указанным адресом, и у
↪  которых поле Linker равно этому адресу.
6. Пройтись по всем сущностями Link, которые ссылаются на сущность Link с указанным адресом, и у
↪  которых поле Target равно этому адресу.

7. Создать сущность Link со значением (смыслом), которым являются адреса на другие 3 сущности
↪  Link (где первая является "начальной", вторая является "связкой", а третья является
↪  "конечной").
8. Обновление сущности Link с указанным адресом новым значением (смыслом), которым являются
↪  адреса на другие 3 сущности Link (где первая является "начальной", вторая является
↪  "связкой", а третья является "конечной").
9. Удаление сущности Link с указаным адресом.
10. Поиск сущности Link со значением (смыслом), которым являются адреса на другие 3 сущности
↪  Link (где первая является "начальной", вторая является "связкой", а третья является
↪  "конечной").
*/
```

1.4 ./csharp/Platform.Data.Triplets/Link.Debug.cs

```
using System;
using System.Diagnostics;

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

namespace Platform.Data.Triplets
{
    public partial struct Link
    {
        #region Properties

        // ReSharper disable InconsistentNaming
        // ReSharper disable UnusedMember.Local
```

```
#pragma warning disable IDE0051 // Remove unused private members

        [DebuggerDisplay(null, Name = "Source")]
        private Link Я_A => this == null ? Itself : Source;

        [DebuggerDisplay(null, Name = "Linker")]
        private Link Я_B => this == null ? Itself : Linker;

        [DebuggerDisplay(null, Name = "Target")]
        private Link Я_C => this == null ? Itself : Target;

        [DebuggerDisplay("Count = {Я_DC}", Name = "ReferersBySource")]
        private Link[] Я_D => this.GetArrayOfRererersBySource();

        [DebuggerDisplay("Count = {Я_EC}", Name = "ReferersByLinker")]
        private Link[] Я_E => this.GetArrayOfRererersByLinker();

        [DebuggerDisplay("Count = {Я_FC}", Name = "ReferersByTarget")]
        private Link[] Я_F => this.GetArrayOfRererersByTarget();

        [DebuggerBrowsable(DebuggerBrowsableState.Never)]
        private Int64 Я_DC => this == null ? 0 : ReferersBySourceCount;

        [DebuggerBrowsable(DebuggerBrowsableState.Never)]
        private Int64 Я_EC => this == null ? 0 : ReferersByLinkerCount;

        [DebuggerBrowsable(DebuggerBrowsableState.Never)]
        private Int64 Я_FC => this == null ? 0 : ReferersByTargetCount;

        [DebuggerDisplay(null, Name = "Timestamp")]
        private DateTime Я_H => this == null ? DateTime.MinValue : Timestamp;

        // ReSharper restore UnusedMember.Local
        // ReSharper restore InconsistentNaming
#pragma warning restore IDE0051 // Remove unused private members

        #endregion

        public override string ToString()
        {
            const string nullString = "null";
            if (this == null)
            {
                return nullString;
            }
            else
            {
                if (this.TryGetName(out string name))
                {
                    return name;
                }
                else
                {
                    return ((long)_link).ToString();
                }
            }
        }
    }
}
```

1.5 ./csharp/Platform.Data.Triplets/Link.cs

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Runtime.InteropServices;
using System.Threading;
using Int = System.Int64;
using LinkIndex = System.UInt64;

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

namespace Platform.Data.Triplets
{
    public struct LinkDefinition : IEquatable<LinkDefinition>
    {
        public readonly Link Source;
        public readonly Link Linker;
        public readonly Link Target;

        public LinkDefinition(Link source, Link linker, Link target)
        {
```

```
21              Source = source;
22              Linker = linker;
23              Target = target;
24          }
25
26          public LinkDefinition(Link link) : this(link.Source, link.Linker, link.Target) { }
27
28          public bool Equals(LinkDefinition other) => Source == other.Source && Linker ==
                ↪  other.Linker && Target == other.Target;
29      }
30
31      public partial struct Link : ILink<Link>, IEquatable<Link>
32      {
33          private const string DllName = "Platform_Data_Triplets_Kernel";
34
35          // TODO: Заменить на очередь событий, по примеру Node.js (+сделать выключаемым)
36          public delegate void CreatedDelegate(LinkDefinition createdLink);
37          public static event CreatedDelegate CreatedEvent = createdLink => { };
38
39          public delegate void UpdatedDelegate(LinkDefinition linkBeforeUpdate, LinkDefinition
                ↪  linkAfterUpdate);
40          public static event UpdatedDelegate UpdatedEvent = (linkBeforeUpdate, linkAfterUpdate)
                ↪  => { };
41
42          public delegate void DeletedDelegate(LinkDefinition deletedLink);
43          public static event DeletedDelegate DeletedEvent = deletedLink => { };
44
45          #region Low Level
46
47          #region Basic Operations
48
49          [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
50          private static extern LinkIndex GetSourceIndex(LinkIndex link);
51
52          [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
53          private static extern LinkIndex GetLinkerIndex(LinkIndex link);
54
55          [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
56          private static extern LinkIndex GetTargetIndex(LinkIndex link);
57
58          [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
59          private static extern LinkIndex GetFirstRefererBySourceIndex(LinkIndex link);
60
61          [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
62          private static extern LinkIndex GetFirstRefererByLinkerIndex(LinkIndex link);
63
64          [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
65          private static extern LinkIndex GetFirstRefererByTargetIndex(LinkIndex link);
66
67          [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
68          private static extern Int GetTime(LinkIndex link);
69
70          [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
71          private static extern LinkIndex CreateLink(LinkIndex source, LinkIndex linker, LinkIndex
                ↪  target);
72
73          [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
74          private static extern LinkIndex UpdateLink(LinkIndex link, LinkIndex newSource,
                ↪  LinkIndex newLinker, LinkIndex newTarget);
75
76          [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
77          private static extern void DeleteLink(LinkIndex link);
78
79          [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
80          private static extern LinkIndex ReplaceLink(LinkIndex link, LinkIndex replacement);
81
82          [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
83          private static extern LinkIndex SearchLink(LinkIndex source, LinkIndex linker, LinkIndex
                ↪  target);
84
85          [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
86          private static extern LinkIndex GetMappedLink(Int mappedIndex);
87
88          [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
89          private static extern void SetMappedLink(Int mappedIndex, LinkIndex linkIndex);
90
91          [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
92          private static extern Int OpenLinks(string filename);
93
94          [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
```

```
95          private static extern Int CloseLinks();
96
97          #endregion
98
99          #region Referers Count Selectors
100
101         [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
102         private static extern LinkIndex GetLinkNumberOfReferersBySource(LinkIndex link);
103
104         [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
105         private static extern LinkIndex GetLinkNumberOfReferersByLinker(LinkIndex link);
106
107         [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
108         private static extern LinkIndex GetLinkNumberOfReferersByTarget(LinkIndex link);
109
110         #endregion
111
112         #region Referers Walkers
113
114         private delegate void Visitor(LinkIndex link);
115         private delegate Int StopableVisitor(LinkIndex link);
116
117         [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
118         private static extern void WalkThroughAllReferersBySource(LinkIndex root, Visitor
            ↪  action);
119
120         [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
121         private static extern int WalkThroughReferersBySource(LinkIndex root, StopableVisitor
            ↪  func);
122
123         [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
124         private static extern void WalkThroughAllReferersByLinker(LinkIndex root, Visitor
            ↪  action);
125
126         [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
127         private static extern int WalkThroughReferersByLinker(LinkIndex root, StopableVisitor
            ↪  func);
128
129         [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
130         private static extern void WalkThroughAllReferersByTarget(LinkIndex root, Visitor
            ↪  action);
131
132         [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
133         private static extern int WalkThroughReferersByTarget(LinkIndex root, StopableVisitor
            ↪  func);
134
135         [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
136         private static extern void WalkThroughAllLinks(Visitor action);
137
138         [DllImport(DllName, CallingConvention = CallingConvention.Cdecl)]
139         private static extern int WalkThroughLinks(StopableVisitor func);
140
141         #endregion
142
143         #endregion
144
145         #region Constains
146
147         public static readonly Link Itself = null;
148         public static readonly bool Continue = true;
149         public static readonly bool Stop;
150
151         #endregion
152
153         #region Static Fields
154
155         private static readonly object _lockObject = new object();
156         private static volatile int _memoryManagerIsReady;
157         private static readonly Dictionary<ulong, long> _linkToMappingIndex = new
            ↪  Dictionary<ulong, long>();
158
159         #endregion
160
161         #region Fields
162
163         [DebuggerBrowsable(DebuggerBrowsableState.Never)]
164         private readonly LinkIndex _link;
165
166         #endregion
167
168         #region Properties
```

```
169
170         [DebuggerBrowsable(DebuggerBrowsableState.Never)]
171         public Link Source => GetSourceIndex(_link);
172
173         [DebuggerBrowsable(DebuggerBrowsableState.Never)]
174         public Link Linker => GetLinkerIndex(_link);
175
176         [DebuggerBrowsable(DebuggerBrowsableState.Never)]
177         public Link Target => GetTargetIndex(_link);
178
179         [DebuggerBrowsable(DebuggerBrowsableState.Never)]
180         public Link FirstRefererBySource => GetFirstRefererBySourceIndex(_link);
181
182         [DebuggerBrowsable(DebuggerBrowsableState.Never)]
183         public Link FirstRefererByLinker => GetFirstRefererByLinkerIndex(_link);
184
185         [DebuggerBrowsable(DebuggerBrowsableState.Never)]
186         public Link FirstRefererByTarget => GetFirstRefererByTargetIndex(_link);
187
188         [DebuggerBrowsable(DebuggerBrowsableState.Never)]
189         public Int RefererBySourceCount => (Int)GetLinkNumberOfRefererBySource(_link);
190
191         [DebuggerBrowsable(DebuggerBrowsableState.Never)]
192         public Int RefererByLinkerCount => (Int)GetLinkNumberOfRefererByLinker(_link);
193
194         [DebuggerBrowsable(DebuggerBrowsableState.Never)]
195         public Int RefererByTargetCount => (Int)GetLinkNumberOfRefererByTarget(_link);
196
197         [DebuggerBrowsable(DebuggerBrowsableState.Never)]
198         public Int TotalReferer => (Int)GetLinkNumberOfRefererBySource(_link) +
        ↪   (Int)GetLinkNumberOfRefererByLinker(_link) +
        ↪   (Int)GetLinkNumberOfRefererByTarget(_link);
199
200         [DebuggerBrowsable(DebuggerBrowsableState.Never)]
201         public DateTime Timestamp => DateTime.FromFileTimeUtc(GetTime(_link));
202
203         #endregion
204
205         #region Infrastructure
206
207         public Link(LinkIndex link) => _link = link;
208
209         public static void StartMemoryManager(string storageFilename)
210         {
211             lock (_lockObject)
212             {
213                 if (_memoryManagerIsReady == default)
214                 {
215                     if (OpenLinks(storageFilename) == 0)
216                     {
217                         throw new InvalidOperationException($"Файл ({storageFilename})
                        ↪   хранилища не удалось открыть.");
218                     }
219                     Interlocked.Exchange(ref _memoryManagerIsReady, 1);
220                 }
221             }
222         }
223
224         public static void StopMemoryManager()
225         {
226             lock (_lockObject)
227             {
228                 if (_memoryManagerIsReady != default)
229                 {
230                     if (CloseLinks() == 0)
231                     {
232                         throw new InvalidOperationException("Файл хранилища не удалось закрыть.
                        ↪   Возможно он был уже закрыт, или не открывался вовсе.");
233                     }
234                     Interlocked.Exchange(ref _memoryManagerIsReady, 0);
235                 }
236             }
237         }
238
239         public static implicit operator LinkIndex?(Link link) => link._link == 0 ?
        ↪   (LinkIndex?)null : link._link;
240
241         public static implicit operator Link(LinkIndex? link) => new Link(link ?? 0);
242
```

```
243         public static implicit operator Int(Link link) => (Int)link._link;
244
245         public static implicit operator Link(Int link) => new Link((LinkIndex)link);
246
247         public static implicit operator LinkIndex(Link link) => link._link;
248
249         public static implicit operator Link(LinkIndex link) => new Link(link);
250
251         public static explicit operator Link(List<Link> links) => LinkConverter.FromList(links);
252
253         public static explicit operator Link(Link[] links) => LinkConverter.FromList(links);
254
255         public static explicit operator Link(string @string) =>
            ↪ LinkConverter.FromString(@string);
256
257         public static bool operator ==(Link first, Link second) => first.Equals(second);
258
259         public static bool operator !=(Link first, Link second) => !first.Equals(second);
260
261         public static Link operator &(Link first, Link second) => Create(first, Net.And, second);
262
263         public override bool Equals(object obj) => obj is Link link ? Equals(link) : false;
264
265         public bool Equals(Link other) => _link == other._link || (LinkDoesNotExist(_link) &&
            ↪ LinkDoesNotExist(other._link));
266
267         public override int GetHashCode() => base.GetHashCode();
268
269         private static bool LinkDoesNotExist(LinkIndex link) => link == 0 ||
            ↪ GetLinkerIndex(link) == 0;
270
271         private static bool LinkWasDeleted(LinkIndex link) => link != 0 && GetLinkerIndex(link)
            ↪ == 0;
272
273         private bool IsMatchingTo(Link source, Link linker, Link target)
274             => ((Source == this && source == null) || (Source == source))
275             && ((Linker == this && linker == null) || (Linker == linker))
276             && ((Target == this && target == null) || (Target == target));
277
278         public LinkIndex ToIndex() => _link;
279
280         public Int ToInt() => (Int)_link;
281
282         #endregion
283
284         #region Basic Operations
285
286         public static Link Create(Link source, Link linker, Link target)
287         {
288             if (_memoryManagerIsReady == default)
289             {
290                 throw new InvalidOperationException("Менеджер памяти ещё не готов.");
291             }
292             if (LinkWasDeleted(source))
293             {
294                 throw new ArgumentException("Удалённая связь не может использоваться в качестве
                    ↪ значения.", nameof(source));
295             }
296             if (LinkWasDeleted(linker))
297             {
298                 throw new ArgumentException("Удалённая связь не может использоваться в качестве
                    ↪ значения.", nameof(linker));
299             }
300             if (LinkWasDeleted(target))
301             {
302                 throw new ArgumentException("Удалённая связь не может использоваться в качестве
                    ↪ значения.", nameof(target));
303             }
304             Link link = CreateLink(source, linker, target);
305             if (link == null)
306             {
307                 throw new InvalidOperationException("Невозможно создать связь.");
308             }
309             CreatedEvent.Invoke(new LinkDefinition(link));
310             return link;
311         }
312
313         public static Link Restore(Int index) => Restore((LinkIndex)index);
314
```

```
315         public static Link Restore(LinkIndex index)
316         {
317             if (_memoryManagerIsReady == default)
318             {
319                 throw new InvalidOperationException("Менеджер памяти ещё не готов.");
320             }
321             if (index == 0)
322             {
323                 throw new ArgumentException("У связи не может быть нулевого адреса.");
324             }
325             try
326             {
327                 Link link = index;
328                 if (LinkDoesNotExist(link))
329                 {
330                     throw new InvalidOperationException("Связь с указанным адресом удалена, либо
                        ↪  не существовала.");
331                 }
332                 return link;
333             }
334             catch (Exception ex)
335             {
336                 throw new InvalidOperationException("Указатель не является корректным.", ex);
337             }
338         }
339 
340         public static Link CreateMapped(Link source, Link linker, Link target, object
            ↪  mappingIndex) => CreateMapped(source, linker, target, Convert.ToInt64(mappingIndex));
341 
342         public static Link CreateMapped(Link source, Link linker, Link target, Int mappingIndex)
343         {
344             if (_memoryManagerIsReady == default)
345             {
346                 throw new InvalidOperationException("Менеджер памяти ещё не готов.");
347             }
348             Link mappedLink = GetMappedLink(mappingIndex);
349             if (mappedLink == null)
350             {
351                 mappedLink = Create(source, linker, target);
352                 SetMappedLink(mappingIndex, mappedLink);
353                 if (GetMappedLink(mappingIndex) != mappedLink)
354                 {
355                     throw new InvalidOperationException("Установить привязанную связь не
                        ↪  удалось.");
356                 }
357             }
358             else if (!mappedLink.IsMatchingTo(source, linker, target))
359             {
360                 throw new InvalidOperationException("Существующая привязанная связь не
                    ↪  соответствует указанным Source, Linker и Target.");
361             }
362             _linkToMappingIndex[mappedLink] = mappingIndex;
363             return mappedLink;
364         }
365 
366         public static bool TrySetMapped(Link link, Int mappingIndex, bool rewrite = false)
367         {
368             Link mappedLink = GetMappedLink(mappingIndex);
369 
370             if (mappedLink == null || rewrite)
371             {
372                 mappedLink = link;
373                 SetMappedLink(mappingIndex, mappedLink);
374                 if (GetMappedLink(mappingIndex) != mappedLink)
375                 {
376                     return false;
377                 }
378             }
379             else if (!mappedLink.IsMatchingTo(link.Source, link.Linker, link.Target))
380             {
381                 return false;
382             }
383             _linkToMappingIndex[mappedLink] = mappingIndex;
384             return true;
385         }
386 
387         public static Link GetMapped(object mappingIndex) =>
            ↪  GetMapped(Convert.ToInt64(mappingIndex));
```

```
388
389         public static Link GetMapped(Int mappingIndex)
390         {
391             if (!TryGetMapped(mappingIndex, out Link mappedLink))
392             {
393                 throw new InvalidOperationException($"Mapped link with index {mappingIndex} is
                    ↪  not set.");
394             }
395             return mappedLink;
396         }
397
398         public static Link GetMappedOrDefault(object mappingIndex)
399         {
400             TryGetMapped(mappingIndex, out Link mappedLink);
401             return mappedLink;
402         }
403
404         public static Link GetMappedOrDefault(Int mappingIndex)
405         {
406             TryGetMapped(mappingIndex, out Link mappedLink);
407             return mappedLink;
408         }
409
410         public static bool TryGetMapped(object mappingIndex, out Link mappedLink) =>
            ↪  TryGetMapped(Convert.ToInt64(mappingIndex), out mappedLink);
411
412         public static bool TryGetMapped(Int mappingIndex, out Link mappedLink)
413         {
414             if (_memoryManagerIsReady == default)
415             {
416                 throw new InvalidOperationException("Менеджер памяти ещё не готов.");
417             }
418             mappedLink = GetMappedLink(mappingIndex);
419             if (mappedLink != null)
420             {
421                 _linkToMappingIndex[mappedLink] = mappingIndex;
422             }
423             return mappedLink != null;
424         }
425
426         public static Link Update(Link link, Link newSource, Link newLinker, Link newTarget)
427         {
428             Update(ref link, newSource, newLinker, newTarget);
429             return link;
430         }
431
432         public static void Update(ref Link link, Link newSource, Link newLinker, Link newTarget)
433         {
434             if (_memoryManagerIsReady == default)
435             {
436                 throw new InvalidOperationException("Менеджер памяти ещё не готов.");
437             }
438             if (LinkDoesNotExist(link))
439             {
440                 throw new ArgumentException("Нельзя обновить несуществующую связь.",
                    ↪  nameof(link));
441             }
442             if (LinkWasDeleted(newSource))
443             {
444                 throw new ArgumentException("Удалённая связь не может использоваться в качестве
                    ↪  нового значения.", nameof(newSource));
445             }
446             if (LinkWasDeleted(newLinker))
447             {
448                 throw new ArgumentException("Удалённая связь не может использоваться в качестве
                    ↪  нового значения.", nameof(newLinker));
449             }
450             if (LinkWasDeleted(newTarget))
451             {
452                 throw new ArgumentException("Удалённая связь не может использоваться в качестве
                    ↪  нового значения.", nameof(newTarget));
453             }
454             LinkIndex previousLinkIndex = link;
455             _linkToMappingIndex.TryGetValue(link, out long mappingIndex);
456             var previousDefinition = new LinkDefinition(link);
457             link = UpdateLink(link, newSource, newLinker, newTarget);
458             if (mappingIndex >= 0 && previousLinkIndex != link)
459             {
```

```
460                 _linkToMappingIndex.Remove(previousLinkIndex);
461                 SetMappedLink(mappingIndex, link);
462                 _linkToMappingIndex.Add(link, mappingIndex);
463             }
464             UpdatedEvent(previousDefinition, new LinkDefinition(link));
465         }
466 
467         public static void Delete(Link link) => Delete(ref link);
468 
469         public static void Delete(ref Link link)
470         {
471             if (LinkDoesNotExist(link))
472             {
473                 return;
474             }
475             LinkIndex previousLinkIndex = link;
476             _linkToMappingIndex.TryGetValue(link, out long mappingIndex);
477             var previousDefinition = new LinkDefinition(link);
478             DeleteLink(link);
479             link = null;
480             if (mappingIndex >= 0)
481             {
482                 _linkToMappingIndex.Remove(previousLinkIndex);
483                 SetMappedLink(mappingIndex, 0);
484             }
485             DeletedEvent(previousDefinition);
486         }
487 
488         //public static void Replace(ref Link link, Link replacement)
489         //{
490         //    if (!MemoryManagerIsReady)
491         //        throw new InvalidOperationException("Менеджер памяти ещё не готов.");
492         //    if (LinkDoesNotExist(link))
493         //        throw new InvalidOperationException("Если связь не существует, её нельзя
    ↪  заменить.");
494         //    if (LinkDoesNotExist(replacement))
495         //        throw new ArgumentException("Пустая или удалённая связь не может быть
    ↪  замещаемым значением.", "replacement");
496         //    link = ReplaceLink(link, replacement);
497         //}
498 
499         public static Link Search(Link source, Link linker, Link target)
500         {
501             if (_memoryManagerIsReady == default)
502             {
503                 throw new InvalidOperationException("Менеджер памяти ещё не готов.");
504             }
505             if (LinkDoesNotExist(source) || LinkDoesNotExist(linker) || LinkDoesNotExist(target))
506             {
507                 throw new InvalidOperationException("Выполнить поиск связи можно только по
                    ↪  существующим связям.");
508             }
509             return SearchLink(source, linker, target);
510         }
511 
512         public static bool Exists(Link source, Link linker, Link target) => SearchLink(source,
    ↪  linker, target) != 0;
513 
514         #endregion
515 
516         #region Referers Walkers
517 
518         public bool WalkThroughReferersAsSource(Func<Link, bool> walker)
519         {
520             if (LinkDoesNotExist(this))
521             {
522                 throw new InvalidOperationException("С несуществующей связью нельзя
                    ↪  производитить операции.");
523             }
524             var referers = ReferersBySourceCount;
525             if (referers == 1)
526             {
527                 return walker(FirstRefererBySource);
528             }
529             else if (referers > 1)
530             {
531                 return WalkThroughReferersBySource(this, x => walker(x) ? 1 : 0) != 0;
532             }
```

```
533             else
534             {
535                 return true;
536             }
537         }
538 
539         public void WalkThroughReferersAsSource(Action<Link> walker)
540         {
541             if (LinkDoesNotExist(this))
542             {
543                 throw new InvalidOperationException("С несуществующей связью нельзя
                    ↪  производитить операции.");
544             }
545             var referers = ReferersBySourceCount;
546             if (referers == 1)
547             {
548                 walker(FirstRefererBySource);
549             }
550             else if (referers > 1)
551             {
552                 WalkThroughAllReferersBySource(this, x => walker(x));
553             }
554         }
555 
556         public bool WalkThroughReferersAsLinker(Func<Link, bool> walker)
557         {
558             if (LinkDoesNotExist(this))
559             {
560                 throw new InvalidOperationException("С несуществующей связью нельзя
                    ↪  производитить операции.");
561             }
562             var referers = ReferersByLinkerCount;
563             if (referers == 1)
564             {
565                 return walker(FirstRefererByLinker);
566             }
567             else if (referers > 1)
568             {
569                 return WalkThroughReferersByLinker(this, x => walker(x) ? 1 : 0) != 0;
570             }
571             else
572             {
573                 return true;
574             }
575         }
576 
577         public void WalkThroughReferersAsLinker(Action<Link> walker)
578         {
579             if (LinkDoesNotExist(this))
580             {
581                 throw new InvalidOperationException("С несуществующей связью нельзя
                    ↪  производитить операции.");
582             }
583             var referers = ReferersByLinkerCount;
584             if (referers == 1)
585             {
586                 walker(FirstRefererByLinker);
587             }
588             else if (referers > 1)
589             {
590                 WalkThroughAllReferersByLinker(this, x => walker(x));
591             }
592         }
593 
594         public bool WalkThroughReferersAsTarget(Func<Link, bool> walker)
595         {
596             if (LinkDoesNotExist(this))
597             {
598                 throw new InvalidOperationException("С несуществующей связью нельзя
                    ↪  производитить операции.");
599             }
600             var referers = ReferersByTargetCount;
601             if (referers == 1)
602             {
603                 return walker(FirstRefererByTarget);
604             }
605             else if (referers > 1)
606             {
```

```
607                 return WalkThroughReferersByTarget(this, x => walker(x) ? 1 : 0) != 0;
608             }
609             else
610             {
611                 return true;
612             }
613         }
614 
615         public void WalkThroughReferersAsTarget(Action<Link> walker)
616         {
617             if (LinkDoesNotExist(this))
618             {
619                 throw new InvalidOperationException("С несуществующей связью нельзя
                    ↪  производитить операции.");
620             }
621             var referers = ReferersByTargetCount;
622             if (referers == 1)
623             {
624                 walker(FirstRefererByTarget);
625             }
626             else if (referers > 1)
627             {
628                 WalkThroughAllReferersByTarget(this, x => walker(x));
629             }
630         }
631 
632         public void WalkThroughReferers(Action<Link> walker)
633         {
634             if (LinkDoesNotExist(this))
635             {
636                 throw new InvalidOperationException("С несуществующей связью нельзя
                    ↪  производитить операции.");
637             }
638             void wrapper(ulong x) => walker(x);
639             WalkThroughAllReferersBySource(this, wrapper);
640             WalkThroughAllReferersByLinker(this, wrapper);
641             WalkThroughAllReferersByTarget(this, wrapper);
642         }
643 
644         public void WalkThroughReferers(Func<Link, bool> walker)
645         {
646             if (LinkDoesNotExist(this))
647             {
648                 throw new InvalidOperationException("С несуществующей связью нельзя
                    ↪  производитить операции.");
649             }
650             long wrapper(ulong x) => walker(x) ? 1 : 0;
651             WalkThroughReferersBySource(this, wrapper);
652             WalkThroughReferersByLinker(this, wrapper);
653             WalkThroughReferersByTarget(this, wrapper);
654         }
655 
656         public static bool WalkThroughAllLinks(Func<Link, bool> walker) => WalkThroughLinks(x =>
            ↪  walker(x) ? 1 : 0) != 0;
657 
658         public static void WalkThroughAllLinks(Action<Link> walker) => WalkThroughAllLinks(new
            ↪  Visitor(x => walker(x)));
659 
660         #endregion
661     }
662 }
```

1.6 ./csharp/Platform.Data.Triplets/LinkConverter.cs

```
1  using System;
2  using System.Collections.Generic;
3  using Platform.Data.Sequences;
4 
5  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6 
7  namespace Platform.Data.Triplets
8  {
9      public static class LinkConverter
10     {
11         public static Link FromList(List<Link> links)
12         {
13             var i = links.Count - 1;
14             var element = links[i];
15             while (--i >= 0)
16             {
```

```
            element = links[i] & element;
        }
        return element;
    }

    public static Link FromList(Link[] links)
    {
        var i = links.Length - 1;
        var element = links[i];
        while (--i >= 0)
        {
            element = links[i] & element;
        }
        return element;
    }

    public static List<Link> ToList(Link link)
    {
        var list = new List<Link>();
        SequenceWalker.WalkRight(link, x => x.Source, x => x.Target, x => x.Linker !=
        ↪  Net.And, list.Add);
        return list;
    }

    public static Link FromNumber(long number) => NumberHelpers.FromNumber(number);

    public static long ToNumber(Link number) => NumberHelpers.ToNumber(number);

    public static Link FromChar(char c) => CharacterHelpers.FromChar(c);

    public static char ToChar(Link charLink) => CharacterHelpers.ToChar(charLink);

    public static Link FromChars(char[] chars) => FromObjectsToSequence(chars, FromChar);

    public static Link FromChars(char[] chars, int takeFrom, int takeUntil) =>
    ↪  FromObjectsToSequence(chars, takeFrom, takeUntil, FromChar);

    public static Link FromNumbers(long[] numbers) => FromObjectsToSequence(numbers,
    ↪  FromNumber);

    public static Link FromNumbers(long[] numbers, int takeFrom, int takeUntil) =>
    ↪  FromObjectsToSequence(numbers, takeFrom, takeUntil, FromNumber);

    public static Link FromNumbers(ushort[] numbers) => FromObjectsToSequence(numbers, x =>
    ↪  FromNumber(x));

    public static Link FromNumbers(ushort[] numbers, int takeFrom, int takeUntil) =>
    ↪  FromObjectsToSequence(numbers, takeFrom, takeUntil, x => FromNumber(x));

    public static Link FromNumbers(uint[] numbers) => FromObjectsToSequence(numbers, x =>
    ↪  FromNumber(x));

    public static Link FromNumbers(uint[] numbers, int takeFrom, int takeUntil) =>
    ↪  FromObjectsToSequence(numbers, takeFrom, takeUntil, x => FromNumber(x));

    public static Link FromNumbers(byte[] numbers) => FromObjectsToSequence(numbers, x =>
    ↪  FromNumber(x));

    public static Link FromNumbers(byte[] numbers, int takeFrom, int takeUntil) =>
    ↪  FromObjectsToSequence(numbers, takeFrom, takeUntil, x => FromNumber(x));

    public static Link FromNumbers(bool[] numbers) => FromObjectsToSequence(numbers, x =>
    ↪  FromNumber(x ? 1 : 0));

    public static Link FromNumbers(bool[] numbers, int takeFrom, int takeUntil) =>
    ↪  FromObjectsToSequence(numbers, takeFrom, takeUntil, x => FromNumber(x ? 1 : 0));

    public static Link FromObjectsToSequence<T>(T[] objects, Func<T, Link> converter) =>
    ↪  FromObjectsToSequence(objects, 0, objects.Length, converter);

    public static Link FromObjectsToSequence<T>(T[] objects, int takeFrom, int takeUntil,
    ↪  Func<T, Link> converter)
    {
        var length = takeUntil - takeFrom;
        if (length <= 0)
        {
            throw new ArgumentOutOfRangeException(nameof(takeUntil), "Нельзя преобразовать
            ↪  пустой список к связям.");
```

```
            }
            var copy = new Link[length];
            for (int i = takeFrom, j = 0; i < takeUntil; i++, j++)
            {
                copy[j] = converter(objects[i]);
            }
            return FromList(copy);
        }

        public static Link FromChars(string str)
        {
            var copy = new Link[str.Length];
            for (var i = 0; i < copy.Length; i++)
            {
                copy[i] = FromChar(str[i]);
            }
            return FromList(copy);
        }

        public static Link FromString(string str)
        {
            var copy = new Link[str.Length];
            for (var i = 0; i < copy.Length; i++)
            {
                copy[i] = FromChar(str[i]);
            }
            var strLink = Link.Create(Net.String, Net.ThatConsistsOf, FromList(copy));
            return strLink;
        }

        public static string ToString(Link link)
        {
            if (link.IsString())
            {
                return ToString(ToList(link.Target));
            }
            throw new ArgumentOutOfRangeException(nameof(link), "Specified link is not a
            ↪  string.");
        }

        public static string ToString(List<Link> charLinks)
        {
            var chars = new char[charLinks.Count];
            for (var i = 0; i < charLinks.Count; i++)
            {
                chars[i] = ToChar(charLinks[i]);
            }
            return new string(chars);
        }
    }
}
```

1.7 ./csharp/Platform.Data.Triplets/LinkExtensions.cs

```
using System;
using System.Collections.Generic;
using System.Text;
using Platform.Data.Sequences;
using Platform.Data.Triplets.Sequences;

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

namespace Platform.Data.Triplets
{
    public static class LinkExtensions
    {
        public static Link SetName(this Link link, string name)
        {
            Link.Create(link, Net.Has, Link.Create(Net.Name, Net.ThatIsRepresentedBy,
            ↪  LinkConverter.FromString(name)));
            return link; // Chaining
        }

        private static readonly HashSet<Link> _linksWithNamesGatheringProcess = new
        ↪  HashSet<Link>();

        public static bool TryGetName(this Link link, out string str)
        {
            // Защита от зацикливания
            if (!_linksWithNamesGatheringProcess.Add(link))
```

```
25              {
26                  str = "...";
27                  return true;
28              }
29              try
30              {
31                  if (link != null)
32                  {
33                      if (link.Linker == Net.And)
34                      {
35                          str = SequenceHelpers.FormatSequence(link);
36                          return true;
37                      }
38                      else if (link.IsGroup())
39                      {
40                          str = LinkConverter.ToString(LinkConverter.ToList(link.Target));
41                          return true;
42                      }
43                      else if (link.IsChar())
44                      {
45                          str = LinkConverter.ToChar(link).ToString();
46                          return true;
47                      }
48                      else if (link.TryGetSpecificName(out str))
49                      {
50                          return true;
51                      }
52
53                      if (link.Source == link || link.Linker == link || link.Target == link)
54                      {
55                          return false;
56                      }
57
58                      if (link.Source.TryGetName(out string sourceName) &&
                        ↪  link.Linker.TryGetName(out string linkerName) &&
                        ↪  link.Target.TryGetName(out string targetName))
59                      {
60                          var sb = new StringBuilder();
61                          sb.Append(sourceName).Append(' ').Append(linkerName).Append('
                            ↪  ').Append(targetName);
62                          str = sb.ToString();
63                          return true;
64                      }
65                  }
66                  str = null;
67                  return false;
68              }
69              finally
70              {
71                  _linksWithNamesGatheringProcess.Remove(link);
72              }
73          }
74
75          public static bool TryGetSpecificName(this Link link, out string name)
76          {
77              string nameLocal = null;
78              if (Net.Name.ReferersBySourceCount < link.ReferersBySourceCount)
79              {
80                  Net.Name.WalkThroughReferersAsSource(referer =>
81                  {
82                      if (referer.Linker == Net.ThatIsRepresentedBy)
83                      {
84                          if (Link.Exists(link, Net.Has, referer))
85                          {
86                              nameLocal = LinkConverter.ToString(referer.Target);
87                              return false; // Останавливаем проход
88                          }
89                      }
90                      return true;
91                  });
92              }
93              else
94              {
95                  link.WalkThroughReferersAsSource(referer =>
96                  {
97                      if (referer.Linker == Net.Has)
98                      {
99                          var nameLink = referer.Target;
```

```
                    if (nameLink.Source == Net.Name && nameLink.Linker ==
                    ↪  Net.ThatIsRepresentedBy)
                    {
                        nameLocal = LinkConverter.ToString(nameLink.Target);
                        return false; // Останавливаем проход
                    }
                }
                return true;
            });
        }

        name = nameLocal;
        return nameLocal != null;
    }

    // Проверка на пренадлежность классу
    public static bool Is(this Link link, Link @class)
    {
        if (link.Linker == Net.IsA)
        {
            if (link.Target == @class)
            {
                return true;
            }
            else
            {
                return link.Target.Is(@class);
            }
        }
        return false;
    }

    // Несколько не правильное определение, так выйдет, что любая сумма входящая в диапазон
    ↪  значений char будет символом.
    // Нужно изменить определение чара, идеально: char consists of sum of [8, 64].
    public static bool IsChar(this Link link) => CharacterHelpers.IsChar(link);

    public static bool IsGroup(this Link link) => link != null && link.Source == Net.Group
    ↪  && link.Linker == Net.ThatConsistsOf;

    public static bool IsSum(this Link link) => link != null && link.Source == Net.Sum &&
    ↪  link.Linker == Net.Of;

    public static bool IsString(this Link link) => link != null && link.Source == Net.String
    ↪  && link.Linker == Net.ThatConsistsOf;

    public static bool IsName(this Link link) => link != null && link.Source == Net.Name &&
    ↪  link.Linker == Net.Of;

    public static Link[] GetArrayOfRererersBySource(this Link link)
    {
        if (link == null)
        {
            return new Link[0];
        }
        else
        {
            var array = new Link[link.ReferersBySourceCount];
            var index = 0;
            link.WalkThroughReferersAsSource(referer => array[index++] = referer);
            return array;
        }
    }

    public static Link[] GetArrayOfRererersByLinker(this Link link)
    {
        if (link == null)
        {
            return new Link[0];
        }
        else
        {
            var array = new Link[link.ReferersByLinkerCount];
            var index = 0;
            link.WalkThroughReferersAsLinker(referer => array[index++] = referer);
            return array;
        }
    }

```

```
        public static Link[] GetArrayOfRererersByTarget(this Link link)
        {
            if (link == null)
            {
                return new Link[0];
            }
            else
            {
                var array = new Link[link.ReferersByTargetCount];
                var index = 0;
                link.WalkThroughReferersAsTarget(referer => array[index++] = referer);
                return array;
            }
        }

        public static void WalkThroughSequence(this Link link, Action<Link> action) =>
        ↪ SequenceWalker.WalkRight(link, x => x.Source, x => x.Target, x => x.Linker !=
        ↪ Net.And, action);
    }
}
```

1.8 ./csharp/Platform.Data.Triplets/Net.cs

```
using Platform.Threading;

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

namespace Platform.Data.Triplets
{
    public enum NetMapping : long
    {
        Link,
        Thing,
        IsA,
        IsNotA,

        Of,
        And,
        ThatConsistsOf,
        Has,
        Contains,
        ContainedBy,

        One,
        Zero,

        Sum,
        Character,
        String,
        Name,

        Set,
        Group,

        ParsedFrom,
        ThatIs,
        ThatIsBefore,
        ThatIsBetween,
        ThatIsAfter,
        ThatIsRepresentedBy,
        ThatHas,

        Text,
        Path,
        Content,
        EmptyContent,
        Empty,
        Alphabet,
        Letter,
        Case,
        Upper,
        UpperCase,
        Lower,
        LowerCase,
        Code
    }

    public static class Net
    {
        public static Link Link { get; private set; }
        public static Link Thing { get; private set; }
        public static Link IsA { get; private set; }
        public static Link IsNotA { get; private set; }
```

```
61
62          public static Link Of { get; private set; }
63          public static Link And { get; private set; }
64          public static Link ThatConsistsOf { get; private set; }
65          public static Link Has { get; private set; }
66          public static Link Contains { get; private set; }
67          public static Link ContainedBy { get; private set; }
68
69          public static Link One { get; private set; }
70          public static Link Zero { get; private set; }
71
72          public static Link Sum { get; private set; }
73          public static Link Character { get; private set; }
74          public static Link String { get; private set; }
75          public static Link Name { get; private set; }
76
77          public static Link Set { get; private set; }
78          public static Link Group { get; private set; }
79
80          public static Link ParsedFrom { get; private set; }
81          public static Link ThatIs { get; private set; }
82          public static Link ThatIsBefore { get; private set; }
83          public static Link ThatIsBetween { get; private set; }
84          public static Link ThatIsAfter { get; private set; }
85          public static Link ThatIsRepresentedBy { get; private set; }
86          public static Link ThatHas { get; private set; }
87
88          public static Link Text { get; private set; }
89          public static Link Path { get; private set; }
90          public static Link Content { get; private set; }
91          public static Link EmptyContent { get; private set; }
92          public static Link Empty { get; private set; }
93          public static Link Alphabet { get; private set; }
94          public static Link Letter { get; private set; }
95          public static Link Case { get; private set; }
96          public static Link Upper { get; private set; }
97          public static Link UpperCase { get; private set; }
98          public static Link Lower { get; private set; }
99          public static Link LowerCase { get; private set; }
100         public static Link Code { get; private set; }
101
102         static Net() => Create();
103
104         public static Link CreateThing() => Link.Create(Link.Itself, IsA, Thing);
105
106         public static Link CreateMappedThing(object mapping) => Link.CreateMapped(Link.Itself,
            ↪  IsA, Thing, mapping);
107
108         public static Link CreateLink() => Link.Create(Link.Itself, IsA, Link);
109
110         public static Link CreateMappedLink(object mapping) => Link.CreateMapped(Link.Itself,
            ↪  IsA, Link, mapping);
111
112         public static Link CreateSet() => Link.Create(Link.Itself, IsA, Set);
113
114         private static void Create()
115         {
116             #region Core
117
118             IsA = Link.GetMappedOrDefault(NetMapping.IsA);
119             IsNotA = Link.GetMappedOrDefault(NetMapping.IsNotA);
120             Link = Link.GetMappedOrDefault(NetMapping.Link);
121             Thing = Link.GetMappedOrDefault(NetMapping.Thing);
122
123             if (IsA == null || IsNotA == null || Link == null || Thing == null)
124             {
125                 // Наивная инициализация (Не является корректным объяснением).
126                 IsA = Link.CreateMapped(Link.Itself, Link.Itself, Link.Itself, NetMapping.IsA);
                    ↪  // Стоит переделать в "[x] is a member|instance|element of the class [y]"
127                 IsNotA = Link.CreateMapped(Link.Itself, Link.Itself, IsA, NetMapping.IsNotA);
128                 Link = Link.CreateMapped(Link.Itself, IsA, Link.Itself, NetMapping.Link);
129                 Thing = Link.CreateMapped(Link.Itself, IsNotA, Link, NetMapping.Thing);
130
131                 IsA = Link.Update(IsA, IsA, IsA, Link); // Исключение, позволяющие завершить
                    ↪  систему
132             }
133
134             #endregion
135
```

```
            Of = CreateMappedLink(NetMapping.Of);
            And = CreateMappedLink(NetMapping.And);
            ThatConsistsOf = CreateMappedLink(NetMapping.ThatConsistsOf);
            Has = CreateMappedLink(NetMapping.Has);
            Contains = CreateMappedLink(NetMapping.Contains);
            ContainedBy = CreateMappedLink(NetMapping.ContainedBy);

            One = CreateMappedThing(NetMapping.One);
            Zero = CreateMappedThing(NetMapping.Zero);

            Sum = CreateMappedThing(NetMapping.Sum);
            Character = CreateMappedThing(NetMapping.Character);
            String = CreateMappedThing(NetMapping.String);
            Name = Link.CreateMapped(Link.Itself, IsA, String, NetMapping.Name);

            Set = CreateMappedThing(NetMapping.Set);
            Group = CreateMappedThing(NetMapping.Group);

            ParsedFrom = CreateMappedLink(NetMapping.ParsedFrom);
            ThatIs = CreateMappedLink(NetMapping.ThatIs);
            ThatIsBefore = CreateMappedLink(NetMapping.ThatIsBefore);
            ThatIsAfter = CreateMappedLink(NetMapping.ThatIsAfter);
            ThatIsBetween = CreateMappedLink(NetMapping.ThatIsBetween);
            ThatIsRepresentedBy = CreateMappedLink(NetMapping.ThatIsRepresentedBy);
            ThatHas = CreateMappedLink(NetMapping.ThatHas);

            Text = CreateMappedThing(NetMapping.Text);
            Path = CreateMappedThing(NetMapping.Path);
            Content = CreateMappedThing(NetMapping.Content);
            Empty = CreateMappedThing(NetMapping.Empty);
            EmptyContent = Link.CreateMapped(Content, ThatIs, Empty, NetMapping.EmptyContent);
            Alphabet = CreateMappedThing(NetMapping.Alphabet);
            Letter = Link.CreateMapped(Link.Itself, IsA, Character, NetMapping.Letter);
            Case = CreateMappedThing(NetMapping.Case);
            Upper = CreateMappedThing(NetMapping.Upper);
            UpperCase = Link.CreateMapped(Case, ThatIs, Upper, NetMapping.UpperCase);
            Lower = CreateMappedThing(NetMapping.Lower);
            LowerCase = Link.CreateMapped(Case, ThatIs, Lower, NetMapping.LowerCase);
            Code = CreateMappedThing(NetMapping.Code);

            SetNames();
        }

        public static void Recreate()
        {
            ThreadHelpers.InvokeWithExtendedMaxStackSize(() => Link.Delete(IsA));
            CharacterHelpers.Recreate();
            Create();
        }

        private static void SetNames()
        {
            Thing.SetName("thing");
            Link.SetName("link");
            IsA.SetName("is a");
            IsNotA.SetName("is not a");

            Of.SetName("of");
            And.SetName("and");
            ThatConsistsOf.SetName("that consists of");
            Has.SetName("has");
            Contains.SetName("contains");
            ContainedBy.SetName("contained by");

            One.SetName("one");
            Zero.SetName("zero");

            Character.SetName("character");
            Sum.SetName("sum");
            String.SetName("string");
            Name.SetName("name");

            Set.SetName("set");
            Group.SetName("group");

            ParsedFrom.SetName("parsed from");
            ThatIs.SetName("that is");
            ThatIsBefore.SetName("that is before");
            ThatIsAfter.SetName("that is after");
```

```
            ThatIsBetween.SetName("that is between");
            ThatIsRepresentedBy.SetName("that is represented by");
            ThatHas.SetName("that has");

            Text.SetName("text");
            Path.SetName("path");
            Content.SetName("content");
            Empty.SetName("empty");
            EmptyContent.SetName("empty content");
            Alphabet.SetName("alphabet");
            Letter.SetName("letter");
            Case.SetName("case");
            Upper.SetName("upper");
            Lower.SetName("lower");
            Code.SetName("code");
        }
    }
}
```

1.9 ./csharp/Platform.Data.Triplets/NumberHelpers.cs

```
using System;
using System.Collections.Generic;
using System.Globalization;
using Platform.Numbers;

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

namespace Platform.Data.Triplets
{
    public static class NumberHelpers
    {
        public static Link[] NumbersToLinks { get; private set; }
        public static Dictionary<Link, long> LinksToNumbers { get; private set; }

        static NumberHelpers() => Create();

        private static void Create()
        {
            NumbersToLinks = new Link[64];
            LinksToNumbers = new Dictionary<Link, long>();
            NumbersToLinks[0] = Net.One;
            LinksToNumbers[Net.One] = 1;
        }

        public static void Recreate() => Create();

        private static Link FromPowerOf2(long powerOf2)
        {
            var result = NumbersToLinks[powerOf2];
            if (result == null)
            {
                var previousPowerOf2Link = NumbersToLinks[powerOf2 - 1];
                if (previousPowerOf2Link == null)
                {
                    previousPowerOf2Link = NumbersToLinks[0];
                    for (var i = 1; i < powerOf2; i++)
                    {
                        if (NumbersToLinks[i] == null)
                        {
                            var numberLink = Link.Create(Net.Sum, Net.Of, previousPowerOf2Link &
                            ↪  previousPowerOf2Link);
                            var num = (long)System.Math.Pow(2, i);
                            NumbersToLinks[i] = numberLink;
                            LinksToNumbers[numberLink] = num;
                            numberLink.SetName(num.ToString(CultureInfo.InvariantCulture));
                        }
                        previousPowerOf2Link = NumbersToLinks[i];
                    }
                }
                result = Link.Create(Net.Sum, Net.Of, previousPowerOf2Link &
                ↪  previousPowerOf2Link);
                var number = (long)System.Math.Pow(2, powerOf2);
                NumbersToLinks[powerOf2] = result;
                LinksToNumbers[result] = number;
                result.SetName(number.ToString(CultureInfo.InvariantCulture));
            }
            return result;
        }

```

```
58          public static Link FromNumber(long number)
59          {
60              if (number == 0)
61              {
62                  return Net.Zero;
63              }
64              if (number == 1)
65              {
66                  return Net.One;
67              }
68              var links = new Link[Bit.Count(number)];
69              if (number >= 0)
70              {
71                  for (long key = 1, powerOf2 = 0, i = 0; key <= number; key *= 2, powerOf2++)
72                  {
73                      if ((number & key) == key)
74                      {
75                          links[i] = FromPowerOf2(powerOf2);
76                          i++;
77                      }
78                  }
79              }
80              else
81              {
82                  throw new NotSupportedException("Negative numbers are not supported yet.");
83              }
84              var sum = Link.Create(Net.Sum, Net.Of, LinkConverter.FromList(links));
85              return sum;
86          }
87
88          public static long ToNumber(Link link)
89          {
90              if (link == Net.Zero)
91              {
92                  return 0;
93              }
94              if (link == Net.One)
95              {
96                  return 1;
97              }
98              if (link.IsSum())
99              {
100                 var numberParts = LinkConverter.ToList(link.Target);
101                 long number = 0;
102                 for (var i = 0; i < numberParts.Count; i++)
103                 {
104                     GoDownAndTakeIt(numberParts[i], out long numberPart);
105                     number += numberPart;
106                 }
107                 return number;
108             }
109             throw new ArgumentOutOfRangeException(nameof(link), "Specified link is not a
                ↪ number.");
110         }
111
112         private static void GoDownAndTakeIt(Link link, out long number)
113         {
114             if (!LinksToNumbers.TryGetValue(link, out number))
115             {
116                 var previousNumberLink = link.Target.Source;
117                 GoDownAndTakeIt(previousNumberLink, out number);
118                 var previousNumberIndex = (int)System.Math.Log(number, 2);
119                 var newNumberIndex = previousNumberIndex + 1;
120                 var newNumberLink = Link.Create(Net.Sum, Net.Of, previousNumberLink &
                    ↪ previousNumberLink);
121                 number += number;
122                 NumbersToLinks[newNumberIndex] = newNumberLink;
123                 LinksToNumbers[newNumberLink] = number;
124             }
125         }
126     }
127 }
```

1.10 ./csharp/Platform.Data.Triplets/Sequences/CompressionExperiments.cs

```
1  using System;
2  using System.Collections.Generic;
3
4  namespace Platform.Data.Triplets.Sequences
5  {
```

```
6       internal static class CompressionExperiments
7       {
8           public static void RightJoin(ref Link subject, Link @object)
9           {
10              if (subject.Linker == Net.And && subject.ReferersBySourceCount == 0 &&
                ↪  subject.ReferersByTargetCount == 0)
11              {
12                  var subJoint = Link.Search(subject.Target, Net.And, @object);
13                  if (subJoint != null && subJoint != subject)
14                  {
15                      Link.Update(ref subject, subject.Source, Net.And, subJoint);
16                      return;
17                  }
18              }
19              subject = Link.Create(subject, Net.And, @object);
20          }
21
22          //public static Link RightJoinUnsafe(Link subject, Link @object)
23          //{
24          //    if (subject.Linker == Net.And && subject.ReferersBySourceCount == 0 &&
                ↪  subject.ReferersByTargetCount == 0)
25          //    {
26          //        Link subJoint = Link.Search(subject.Target, Net.And, @object);
27          //        if (subJoint != null && subJoint != subject)
28          //        {
29          //            Link.Update(ref subject, subject.Source, Net.And, subJoint);
30          //            return subject;
31          //        }
32          //    }
33          //    return Link.Create(subject, Net.And, @object);
34          //}
35
36          ////public static void LeftJoin(ref Link subject, Link @object)
37          ////{
38          ////    if (subject.Linker == Net.And && subject.ReferersBySourceCount == 0 &&
                ↪  subject.ReferersByTargetCount == 0)
39          ////    {
40          ////        Link subJoint = Link.Search(@object, Net.And, subject.Source);
41          ////        if (subJoint != null && subJoint != subject)
42          ////        {
43          ////            Link.Update(ref subject, subJoint, Net.And, subject.Target);
44          ////            return;
45          ////        }
46          ////    }
47          ////    subject = Link.Create(@object, Net.And, subject);
48          ////}
49
50          public static void LeftJoin(ref Link subject, Link @object)
51          {
52              if (subject.Linker == Net.And && subject.ReferersBySourceCount == 0 &&
                ↪  subject.ReferersByTargetCount == 0)
53              {
54                  var subJoint = Link.Search(@object, Net.And, subject.Source);
55                  if (subJoint != null && subJoint != subject)
56                  {
57                      Link.Update(ref subject, subJoint, Net.And, subject.Target);
58                      //var prev = Link.Search(@object, Net.And, subject);
59                      //if (prev != null)
60                      //{
61                      //    Link.Update(ref prev, subJoint, Net.And, subject.Target);
62                      //}
63                      return;
64                  }
65              }
66              subject = Link.Create(@object, Net.And, subject);
67          }
68
69          // Сначала сжатие налево, а затем направо (так эффективнее)
70          // Не приятный момент, что обе связи, и первая и вторая могут быть изменены в результате
                ↪  алгоритма.
71          //public static Link CombinedJoin(ref Link first, ref Link second)
72          //{
73          //    Link atomicConnection = Link.Search(first, Net.And, second);
74          //    if (atomicConnection != null)
75          //    {
76          //        return atomicConnection;
77          //    }
78          //    else
```

```
79          //    {
80          //        if (second.Linker == Net.And)
81          //        {
82          //            Link subJoint = Link.Search(first, Net.And, second.Source);
83          //            if (subJoint != null && subJoint != second)// && subJoint.TotalReferers >
            ↪  second.TotalReferers)
84          //            {
85          //                //if (first.Linker == Net.And)
86          //                //{
87          //                //    // TODO: ...
88          //                //}
89          //                if (second.TotalReferers > 0)
90          //                {
91          //                    // В данный момент это никак не влияет, из-за того что добавлено
            ↪  условие по требованию
92          //                    // использования атомарного соедининеия если оно есть
93          //
94          //                    // В целом же приоритет между обходным соединением и атомарным
            ↪  нужно определять по весу.
95          //                    // И если в сети обнаружено сразу два варианта прохода - простой и
            ↪  обходной - нужно перебрасывать
96          //                    // пути с меньшим весом на использование путей с большим весом.
            ↪  (Это и технически эффективнее и более оправдано
97          //                    // с точки зрения смысла).
98          //
99          //                    // Положительный эффект текущей реализации, что она быстро
            ↪  "успокаивается" набирает критическую массу
100         //                    // и перестаёт вести себя не предсказуемо
101         //
102         //                    // Неприятность учёта веса в том, что нужно обрабатывать большое
            ↪  количество комбинаций.
103         //                    // Но вероятно это оправдано.
104         //
105         //                    //var prev = Link.Search(first, Net.And, second);
106         //                    //if (prev != null && subJoint != prev) // && prev.TotalReferers <
            ↪  subJoint.TotalReferers)
107         //                    //{
108         //                    //    Link.Update(ref prev, subJoint, Net.And, second.Target);
109         //                    //    if (second.TotalReferers == 0)
110         //                    //    {
111         //                    //        Link.Delete(ref second);
112         //                    //    }
113         //                    //    return prev;
114         //                    //}
115         //                    //return Link.Create(subJoint, Net.And, second.Target);
116         //                }
117         //                else
118         //                {
119         //                    Link.Update(ref second, subJoint, Net.And, second.Target);
120         //                    return second;
121         //                }
122         //            }
123         //        }
124         //        if (first.Linker == Net.And)
125         //        {
126         //            Link subJoint = Link.Search(first.Target, Net.And, second);
127         //            if (subJoint != null && subJoint != first)// && subJoint.TotalReferers >
            ↪  first.TotalReferers)
128         //            {
129         //                if (first.TotalReferers > 0)
130         //                {
131         //                    //var prev = Link.Search(first, Net.And, second);
132         //                    //if (prev != null && subJoint != prev) // && prev.TotalReferers <
            ↪  subJoint.TotalReferers)
133         //                    //{
134         //                    //    Link.Update(ref prev, first.Source, Net.And, subJoint);
135         //                    //    if (first.TotalReferers == 0)
136         //                    //    {
137         //                    //        Link.Delete(ref first);
138         //                    //    }
139         //                    //    return prev;
140         //                    //}
141         //                    //return Link.Create(first.Source, Net.And, subJoint);
142         //                }
143         //                else
144         //                {
145         //                    Link.Update(ref first, first.Source, Net.And, subJoint);
```

```
146         //                    return first;
147         //                }
148         //            }
149         //        }
150         //        return Link.Create(first, Net.And, second);
151         //    }
152         //}
153
154         public static int CompressionsCount;
155
156         public static Link CombinedJoin(ref Link first, ref Link second)
157         {
158             // Перестроение работает хорошо только когда одна из связей является парой и
                ↪  аккумулятором одновременно
159             // Когда обе связи - пары - нужно использовать другой алгоритм, иначе сжатие будет
                ↪  отсутствовать.
160             //if ((first.Linker == Net.And && second.Linker != Net.And)
161             // || (second.Linker == Net.And && first.Linker != Net.And))
162             //{
163             //    Link connection = TryReconstructConnection(first, second);
164             //    if (connection != null)
165             //    {
166             //        CompressionsCount++;
167             //        return connection;
168             //    }
169             //}
170             //return first & second;
171             //long totalDoublets = Net.And.ReferersByLinkerCount;
172             if (first == null || second == null)
173             {
174             }
175             var directConnection = Link.Search(first, Net.And, second);
176             if (directConnection == null)
177             {
178                 directConnection = TryReconstructConnection(first, second);
179             }
180             Link rightCrossConnection = null;
181             if (second.Linker == Net.And)
182             {
183                 var assumedRightCrossConnection = Link.Search(first, Net.And, second.Source);
184                 if (assumedRightCrossConnection != null && second != assumedRightCrossConnection)
185                 {
186                     rightCrossConnection = assumedRightCrossConnection;
187                 }
188                 else
189                 {
190                     rightCrossConnection = TryReconstructConnection(first, second.Source);
191                 }
192             }
193             Link leftCrossConnection = null;
194             if (first.Linker == Net.And)
195             {
196                 var assumedLeftCrossConnection = Link.Search(first.Target, Net.And, second);
197                 if (assumedLeftCrossConnection != null && first != assumedLeftCrossConnection)
198                 {
199                     leftCrossConnection = assumedLeftCrossConnection;
200                 }
201                 else
202                 {
203                     leftCrossConnection = TryReconstructConnection(first.Target, second);
204                 }
205             }
206             // Наверное имеет смысл только в "безвыходной" ситуации
207             //if (directConnection == null && rightCrossConnection == null &&
                ↪  leftCrossConnection == null)
208             //{
209             //    directConnection = TryReconstructConnection(first, second);
210             //    // Может давать более агрессивное сжатие, но теряется стабильность
211             //    //if (directConnection == null)
212             //    //{
213             //    //    //if (second.Linker == Net.And)
214             //    //    //{
215             //    //    //    Link assumedRightCrossConnection = TryReconstructConnection(first,
                ↪  second.Source);
216             //    //    //    if (assumedRightCrossConnection != null && second !=
                ↪  assumedRightCrossConnection)
217             //    //    //    {
218             //    //    //        rightCrossConnection = assumedRightCrossConnection;
```

```
219                 //    //    //    }
220                 //    //    //}
221                 //    //    //if (rightCrossConnection == null)
222                 //    //    //{
223                 //    //    //if (first.Linker == Net.And)
224                 //    //    //{
225                 //    //    //    Link assumedLeftCrossConnection =
                    ↪  TryReconstructConnection(first.Target, second);
226                 //    //    //    if (assumedLeftCrossConnection != null && first !=
                    ↪  assumedLeftCrossConnection)
227                 //    //    //    {
228                 //    //    //        leftCrossConnection = assumedLeftCrossConnection;
229                 //    //    //    }
230                 //    //    //}
231                 //    //    //}
232                 //    //}
233                 //}
234                 //Link middleCrossConnection = null;
235                 //if (second.Linker == Net.And && first.Linker == Net.And)
236                 //{
237                 //    Link assumedMiddleCrossConnection = Link.Search(first.Target, Net.And,
                    ↪  second.Source);
238                 //    if (assumedMiddleCrossConnection != null && first !=
                    ↪  assumedMiddleCrossConnection && second != assumedMiddleCrossConnection)
239                 //    {
240                 //        middleCrossConnection = assumedMiddleCrossConnection;
241                 //    }
242                 //}
243                 //Link rightMiddleCrossConnectinon = null;
244                 //if (middleCrossConnection != null)
245                 //{
246                 //}
247                 if (directConnection != null
248                 && (rightCrossConnection == null || directConnection.TotalReferers >=
                    ↪  rightCrossConnection.TotalReferers)
249                 && (leftCrossConnection == null || directConnection.TotalReferers >=
                    ↪  leftCrossConnection.TotalReferers))
250                 {
251                     if (rightCrossConnection != null)
252                     {
253                         var prev = Link.Search(rightCrossConnection, Net.And, second.Target);
254                         if (prev != null && directConnection != prev)
255                         {
256                             Link.Update(ref prev, first, Net.And, second);
257                         }
258                         if (rightCrossConnection.TotalReferers == 0)
259                         {
260                             Link.Delete(ref rightCrossConnection);
261                         }
262                     }
263                     if (leftCrossConnection != null)
264                     {
265                         var prev = Link.Search(first.Source, Net.And, leftCrossConnection);
266                         if (prev != null && directConnection != prev)
267                         {
268                             Link.Update(ref prev, first, Net.And, second);
269                         }
270                         if (leftCrossConnection.TotalReferers == 0)
271                         {
272                             Link.Delete(ref leftCrossConnection);
273                         }
274                     }
275                     TryReconstructConnection(first, second);
276                     return directConnection;
277                 }
278                 else if (rightCrossConnection != null
279                      && (directConnection == null || rightCrossConnection.TotalReferers >=
                         ↪  directConnection.TotalReferers)
280                      && (leftCrossConnection == null || rightCrossConnection.TotalReferers >=
                         ↪  leftCrossConnection.TotalReferers))
281                 {
282                     if (directConnection != null)
283                     {
284                         var prev = Link.Search(first, Net.And, second);
285                         if (prev != null && rightCrossConnection != prev)
286                         {
287                             Link.Update(ref prev, rightCrossConnection, Net.And, second.Target);
```

```
                }
            }
            if (leftCrossConnection != null)
            {
                var prev = Link.Search(first.Source, Net.And, leftCrossConnection);
                if (prev != null && rightCrossConnection != prev)
                {
                    Link.Update(ref prev, rightCrossConnection, Net.And, second.Target);
                }
            }
            //TryReconstructConnection(first, second.Source);
            //TryReconstructConnection(rightCrossConnection, second.Target); // ухудшает
            ↪ стабильность
            var resultConnection = rightCrossConnection & second.Target;
            //if (second.TotalReferers == 0)
            //    Link.Delete(ref second);
            return resultConnection;
        }
        else if (leftCrossConnection != null
             && (directConnection == null || leftCrossConnection.TotalReferers >=
             ↪ directConnection.TotalReferers)
             && (rightCrossConnection == null || leftCrossConnection.TotalReferers >=
             ↪ rightCrossConnection.TotalReferers))
        {
            if (directConnection != null)
            {
                var prev = Link.Search(first, Net.And, second);
                if (prev != null && leftCrossConnection != prev)
                {
                    Link.Update(ref prev, first.Source, Net.And, leftCrossConnection);
                }
            }
            if (rightCrossConnection != null)
            {
                var prev = Link.Search(rightCrossConnection, Net.And, second.Target);
                if (prev != null && rightCrossConnection != prev)
                {
                    Link.Update(ref prev, first.Source, Net.And, leftCrossConnection);
                }
            }
            //TryReconstructConnection(first.Target, second);
            //TryReconstructConnection(first.Source, leftCrossConnection); // ухудшает
            ↪ стабильность
            var resultConnection = first.Source & leftCrossConnection;
            //if (first.TotalReferers == 0)
            //    Link.Delete(ref first);
            return resultConnection;
        }
        else
        {
            if (directConnection != null)
            {
                return directConnection;
            }
            if (rightCrossConnection != null)
            {
                return rightCrossConnection & second.Target;
            }
            if (leftCrossConnection != null)
            {
                return first.Source & leftCrossConnection;
            }
        }
        // Можно фиксировать по окончанию каждой из веток, какой эффект от неё происходит
        ↪ (на сколько уменьшается/увеличивается количество связей)
        directConnection = first & second;
        //long difference = Net.And.ReferersByLinkerCount - totalDoublets;
        //if (difference != 1)
        //{
        //    Console.WriteLine(Net.And.ReferersByLinkerCount - totalDoublets);
        //}
        return directConnection;
    }

    private static Link TryReconstructConnection(Link first, Link second)
    {
        Link directConnection = null;
        if (second.ReferersBySourceCount < first.ReferersBySourceCount)
```

```
            {
                //  o_|      x_o ...
                // x_|        |___|
                //
                // <-
                second.WalkThroughReferersAsSource(couple =>
                {
                    if (couple.Linker == Net.And && couple.ReferersByTargetCount == 1 &&
                    ↪  couple.ReferersBySourceCount == 0)
                    {
                        var neighbour = couple.FirstRefererByTarget;
                        if (neighbour.Linker == Net.And && neighbour.Source == first)
                        {
                            if (directConnection == null)
                            {
                                directConnection = first & second;
                            }
                            Link.Update(ref neighbour, directConnection, Net.And, couple.Target);
                            //Link.Delete(ref couple); // Можно заменить удалением couple
                        }
                    }
                    if (couple.Linker == Net.And)
                    {
                        var neighbour = couple.FirstRefererByTarget;
                        if (neighbour.Linker == Net.And && neighbour.Source == first)
                        {
                            throw new NotImplementedException();
                        }
                    }
                });
            }
            else
            {
                //  o_|     x_o ...
                // x_|       |___|
                //
                // ->
                first.WalkThroughReferersAsSource(couple =>
                {
                    if (couple.Linker == Net.And)
                    {
                        var neighbour = couple.Target;
                        if (neighbour.Linker == Net.And && neighbour.Source == second)
                        {
                            if (neighbour.ReferersByTargetCount == 1 &&
                            ↪  neighbour.ReferersBySourceCount == 0)
                            {
                                if (directConnection == null)
                                {
                                    directConnection = first & second;
                                }
                                Link.Update(ref couple, directConnection, Net.And,
                                ↪  neighbour.Target);
                                //Link.Delete(ref neighbour);
                            }
                        }
                    }
                });
            }

            if (second.ReferersByTargetCount < first.ReferersByTargetCount)
            {
                // |_x      ... x_o
                //  |_o      |___|
                //
                //   <-
                second.WalkThroughReferersAsTarget(couple =>
                {
                    if (couple.Linker == Net.And)
                    {
                        var neighbour = couple.Source;
                        if (neighbour.Linker == Net.And && neighbour.Target == first)
                        {
                            if (neighbour.ReferersByTargetCount == 0 &&
                            ↪  neighbour.ReferersBySourceCount == 1)
                            {
                                if (directConnection == null)
                                {
```

```
                                    directConnection = first & second;
                                }
                                Link.Update(ref couple, neighbour.Source, Net.And,
                                ↪  directConnection);
                                //Link.Delete(ref neighbour);
                            }
                        }
                    }
                });
            }
            else
            {
                // |_x      ... x_o
                //  |_o      |___|
                //
                //   ->
                first.WalkThroughReferersAsTarget((couple) =>
                {
                    if (couple.Linker == Net.And && couple.ReferersByTargetCount == 0 &&
                    ↪  couple.ReferersBySourceCount == 1)
                    {
                        var neighbour = couple.FirstRefererBySource;
                        if (neighbour.Linker == Net.And && neighbour.Target == second)
                        {
                            if (directConnection == null)
                            {
                                directConnection = first & second;
                            }
                            Link.Update(ref neighbour, couple.Source, Net.And, directConnection);
                            Link.Delete(ref couple);
                        }
                    }
                });
            }
            if (directConnection != null)
            {
                CompressionsCount++;
            }
            return directConnection;
        }

        ////public static Link CombinedJoin(Link left, Link right)
        ////{
        ////    Link rightSubJoint = Link.Search(left, Net.And, right.Source);
        ////    if (rightSubJoint != null && rightSubJoint != right)
        ////    {
        ////        long rightSubJointReferers = rightSubJoint.TotalReferers;
        ////        Link leftSubJoint = Link.Search(left.Target, Net.And, right);
        ////        if (leftSubJoint != null && leftSubJoint != left)
        ////        {
        ////            long leftSubJointReferers = leftSubJoint.TotalReferers;
        ////            if (leftSubJointReferers > rightSubJointReferers)
        ////            {
        ////                long leftReferers = left.TotalReferers;
        ////                if (leftReferers > 0)
        ////                {
        ////                    return Link.Create(left.Source, Net.And, leftSubJoint);
        ////                }
        ////                else
        ////                {
        ////                    Link.Update(ref left, left.Source, Net.And, leftSubJoint);
        ////                    return left;
        ////                }
        ////            }
        ////        }
        ////        long rightReferers = right.TotalReferers;
        ////        if (rightReferers > 0)
        ////        {
        ////            return Link.Create(rightSubJoint, Net.And, right.Target);
        ////        }
        ////        else
        ////        {
        ////            Link.Update(ref right, rightSubJoint, Net.And, right.Target);
        ////            return right;
        ////        }
        ////    }
        ////    return Link.Create(left, Net.And, right);
        ////}
```

```
511         //public static Link CombinedJoin(Link left, Link right)
512         //{
513         //    long leftReferers = left.TotalReferers;
514         //    Link leftSubJoint = Link.Search(left.Target, Net.And, right);
515         //    if (leftSubJoint != null && leftSubJoint != left)
516         //    {
517         //        long leftSubJointReferers = leftSubJoint.TotalReferers;
518         //    }
519         //    long rightReferers = left.TotalReferers;
520         //    Link rightSubJoint = Link.Search(left, Net.And, right.Source);
521         //    long rightSubJointReferers = rightSubJoint != null ? rightSubJoint.TotalReferers :
            ↪  long.MinValue;
522         //}
523         //public static Link LeftJoinUnsafe(Link subject, Link @object)
524         //{
525         //    if (subject.Linker == Net.And && subject.ReferersBySourceCount == 0 &&
            ↪  subject.ReferersByTargetCount == 0)
526         //    {
527         //        Link subJoint = Link.Search(@object, Net.And, subject.Source);
528         //        if (subJoint != null && subJoint != subject)
529         //        {
530         //            Link.Update(ref subject, subJoint, Net.And, subject.Target);
531         //            return subject;
532         //        }
533         //    }
534         //    return Link.Create(@object, Net.And, subject);
535         //}
536
537         public static int ChunkSize = 2;
538
539         //public static Link FromList(List<Link> links)
540         //{
541         //    Link element = links[0];
542         //    for (int i = 1; i < links.Count; i += ChunkSize)
543         //    {
544         //        int j = (i + ChunkSize - 1);
545         //        j = j < links.Count ? j : (links.Count - 1);
546         //        Link subElement = links[j];
547         //        while (--j >= i) LeftJoin(ref subElement, links[j]);
548         //        RightJoin(ref element, subElement);
549         //    }
550         //    return element;
551         //}
552
553         //public static Link FromList(Link[] links)
554         //{
555         //    Link element = links[0];
556         //    for (int i = 1; i < links.Length; i += ChunkSize)
557         //    {
558         //        int j = (i + ChunkSize - 1);
559         //        j = j < links.Length ? j : (links.Length - 1);
560         //        Link subElement = links[j];
561         //        while (--j >= i) LeftJoin(ref subElement, links[j]);
562         //        RightJoin(ref element, subElement);
563         //    }
564         //    return element;
565         //}
566
567         //public static Link FromList(IList<Link> links)
568         //{
569         //    Link element = links[0];
570         //    for (int i = 1; i < links.Count; i += ChunkSize)
571         //    {
572         //        int j = (i + ChunkSize - 1);
573         //        j = j < links.Count ? j : (links.Count - 1);
574         //        Link subElement = links[j];
575         //        while (--j >= i)
576         //        {
577         //            Link x = links[j];
578         //            subElement = CombinedJoin(ref x, ref subElement);
579         //        }
580         //        element = CombinedJoin(ref element, ref subElement);
581         //    }
582         //    return element;
583         //}
584         //public static Link FromList(IList<Link> links)
585         //{
586         //    int i = 0;
```

```
587             //    Link element = links[i++];
588             //    if (links.Count % 2 == 0)
589             //    {
590             //        element = CombinedJoin(element, links[i++]);
591             //    }
592             //    for (; i < links.Count; i += 2)
593             //    {
594             //        Link doublet = CombinedJoin(links[i], links[i + 1]);
595             //        element = CombinedJoin(ref element, ref doublet);
596             //    }
597             //    return element;
598             //}
599
600             // Заглушка, возможно опасная
601             private static Link CombinedJoin(Link element, Link link)
602             {
603                 return CombinedJoin(ref element, ref link);
604             }
605
606             //public static Link FromList(List<Link> links)
607             //{
608             //    int i = links.Count - 1;
609             //    Link element = links[i];
610             //    while (--i >= 0) element = LinkConverterOld.ConnectLinks2(links[i], element,
    ↪  links, ref i);
611             //    return element;
612             //}
613             //public static Link FromList(Link[] links)
614             //{
615             //    int i = links.Length - 1;
616             //    Link element = links[i];
617             //    while (--i >= 0) element = LinkConverterOld.ConnectLinks2(links[i], element,
    ↪  links, ref i);
618             //    return element;
619             //}
620             //public static Link FromList(List<Link> links)
621             //{
622             //    Link element = links[0];
623             //    for (int i = 1; i < links.Count; i += ChunkSize)
624             //    {
625             //        int j = (i + ChunkSize - 1);
626             //        j = j < links.Count ? j : (links.Count - 1);
627             //        Link subElement = links[j];
628             //        while (--j >= i) subElement = CombinedJoin(links[j], subElement);
629             //        element = CombinedJoin(element, subElement);
630             //    }
631             //    return element;
632             //}
633             //public static Link FromList(Link[] links)
634             //{
635             //    Link element = links[0];
636             //    for (int i = 1; i < links.Length; i += ChunkSize)
637             //    {
638             //        int j = (i + ChunkSize - 1);
639             //        j = j < links.Length ? j : (links.Length - 1);
640             //        Link subElement = links[j];
641             //        while (--j >= i) subElement = CombinedJoin(links[j], subElement);
642             //        element = CombinedJoin(element, subElement);
643             //    }
644             //    return element;
645             //}
646             //public static Link FromList(IList<Link> links)
647             //{
648             //    int leftBound = 0;
649             //    int rightBound = links.Count - 1;
650             //    if (leftBound == rightBound)
651             //    {
652             //        return links[0];
653             //    }
654             //    Link left = links[leftBound];
655             //    Link right = links[rightBound];
656             //    long leftReferers = left.ReferersBySourceCount + left.ReferersByTargetCount;
657             //    long rightReferers = right.ReferersBySourceCount + right.ReferersByTargetCount;
658             //    while (true)
659             //    {
660             //        //if (rightBound % 2 != leftBound % 2)
661             //        if (rightReferers >= leftReferers)
662             //        {
```

```
663         //            int nextRightBound = --rightBound;
664         //            if (nextRightBound == leftBound)
665         //            {
666         //                var x = CombinedJoin(ref left, ref right);
667         //                return x;
668         //            }
669         //            else
670         //            {
671         //                Link nextRight = links[nextRightBound];
672         //                right = CombinedJoin(ref nextRight, ref right);
673         //                rightReferers = right.ReferersBySourceCount +
        ↪  right.ReferersByTargetCount;
674         //            }
675         //        }
676         //        else
677         //        {
678         //            int nextLeftBound = ++leftBound;
679         //            if (nextLeftBound == rightBound)
680         //            {
681         //                return CombinedJoin(ref left, ref right);
682         //            }
683         //            else
684         //            {
685         //                Link nextLeft = links[nextLeftBound];
686         //                left = CombinedJoin(ref left, ref nextLeft);
687         //                leftReferers = left.ReferersBySourceCount + left.ReferersByTargetCount;
688         //            }
689         //        }
690         //    }
691         //}
692         //public static Link FromList(IList<Link> links)
693         //{
694         //    int i = links.Count - 1;
695         //    Link element = links[i];
696         //    while (--i >= 0)
697         //    {
698         //        LeftJoin(ref element, links[i]); // LeftJoin(ref element, links[i]);
699         //    }
700         //    return element;
701         //}
702
703         public static Link FromList(List<Link> links)
704         {
705             var i = links.Count - 1;
706             var element = links[i];
707             while (--i >= 0)
708             {
709                 var x = links[i];
710                 element = CombinedJoin(ref x, ref element); // LeftJoin(ref element, links[i]);
711             }
712             return element;
713         }
714
715         public static Link FromList(Link[] links)
716         {
717             var i = links.Length - 1;
718             var element = links[i];
719             while (--i >= 0)
720             {
721                 element = CombinedJoin(ref links[i], ref element); // LeftJoin(ref element,
                    ↪  links[i]);
722             }
723             return element;
724         }
725     }
726 }
```

1.11 ./csharp/Platform.Data.Triplets/Sequences/SequenceHelpers.cs

```
1  using System;
2  using System.Collections.Generic;
3  using System.Text;
4  using Platform.Data.Sequences;
5
6  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8  namespace Platform.Data.Triplets.Sequences
9  {
10     /// <remarks>
11     /// TODO: Check that CollectMatchingSequences algorithm is working, if not throw it away.
```

```
12      /// TODO: Think of the abstraction on Sequences that can be equally usefull for triple
        ↪  links, doublet links and so on.
13      /// </remarks>
14      public static class SequenceHelpers
15      {
16          public static readonly int MaxSequenceFormatSize = 20;
17
18          //public static void DeleteSequence(Link sequence)
19          //{
20          //}
21
22          public static string FormatSequence(Link sequence)
23          {
24              var visitedElements = 0;
25              var sb = new StringBuilder();
26              sb.Append('{');
27              StopableSequenceWalker.WalkRight(sequence, x => x.Source, x => x.Target, x =>
                ↪  x.Linker != Net.And, element =>
28              {
29                  if (visitedElements > 0)
30                  {
31                      sb.Append(',');
32                  }
33                  sb.Append(element.ToString());
34                  visitedElements++;
35                  if (visitedElements < MaxSequenceFormatSize)
36                  {
37                      return true;
38                  }
39                  else
40                  {
41                      sb.Append(", ...");
42                      return false;
43                  }
44              });
45              sb.Append('}');
46              return sb.ToString();
47          }
48
49          public static List<Link> CollectMatchingSequences(Link[] links)
50          {
51              if (links.Length == 1)
52              {
53                  throw new InvalidOperationException("Подпоследовательности с одним элементом не
                    ↪  поддерживаются.");
54              }
55              var leftBound = 0;
56              var rightBound = links.Length - 1;
57              var left = links[leftBound++];
58              var right = links[rightBound--];
59              var results = new List<Link>();
60              CollectMatchingSequences(left, leftBound, links, right, rightBound, ref results);
61              return results;
62          }
63
64          private static void CollectMatchingSequences(Link leftLink, int leftBound, Link[]
            ↪  middleLinks, Link rightLink, int rightBound, ref List<Link> results)
65          {
66              var leftLinkTotalReferers = leftLink.ReferersBySourceCount +
                ↪  leftLink.ReferersByTargetCount;
67              var rightLinkTotalReferers = rightLink.ReferersBySourceCount +
                ↪  rightLink.ReferersByTargetCount;
68              if (leftLinkTotalReferers <= rightLinkTotalReferers)
69              {
70                  var nextLeftLink = middleLinks[leftBound];
71                  var elements = GetRightElements(leftLink, nextLeftLink);
72                  if (leftBound <= rightBound)
73                  {
74                      for (var i = elements.Length - 1; i >= 0; i--)
75                      {
76                          var element = elements[i];
77                          if (element != null)
78                          {
79                              CollectMatchingSequences(element, leftBound + 1, middleLinks,
                                ↪  rightLink, rightBound, ref results);
80                          }
81                      }
82                  }
83                  else
```

```
84                  {
85                      for (var i = elements.Length - 1; i >= 0; i--)
86                      {
87                          var element = elements[i];
88                          if (element != null)
89                          {
90                              results.Add(element);
91                          }
92                      }
93                  }
94              }
95              else
96              {
97                  var nextRightLink = middleLinks[rightBound];
98                  var elements = GetLeftElements(rightLink, nextRightLink);
99                  if (leftBound <= rightBound)
100                 {
101                     for (var i = elements.Length - 1; i >= 0; i--)
102                     {
103                         var element = elements[i];
104                         if (element != null)
105                         {
106                             CollectMatchingSequences(leftLink, leftBound, middleLinks,
                                ↪  elements[i], rightBound - 1, ref results);
107                         }
108                     }
109                 }
110                 else
111                 {
112                     for (var i = elements.Length - 1; i >= 0; i--)
113                     {
114                         var element = elements[i];
115                         if (element != null)
116                         {
117                             results.Add(element);
118                         }
119                     }
120                 }
121             }
122         }
123
124         public static Link[] GetRightElements(Link startLink, Link rightLink)
125         {
126             var result = new Link[4];
127             TryStepRight(startLink, rightLink, result, 0);
128             startLink.WalkThroughReferersAsTarget(couple =>
129                 {
130                     if (couple.Linker == Net.And)
131                     {
132                         if (TryStepRight(couple, rightLink, result, 2))
133                         {
134                             return Link.Stop;
135                         }
136                     }
137                     return Link.Continue;
138                 });
139             return result;
140         }
141
142         public static bool TryStepRight(Link startLink, Link rightLink, Link[] result, int
            ↪  offset)
143         {
144             var added = 0;
145             startLink.WalkThroughReferersAsSource(couple =>
146                 {
147                     if (couple.Linker == Net.And)
148                     {
149                         var coupleTarget = couple.Target;
150                         if (coupleTarget == rightLink)
151                         {
152                             result[offset] = couple;
153                             if (++added == 2)
154                             {
155                                 return Link.Stop;
156                             }
157                         }
158                         else if (coupleTarget.Linker == Net.And && coupleTarget.Source ==
                            ↪  rightLink)
```

```
                        {
                            result[offset + 1] = couple;
                            if (++added == 2)
                            {
                                return Link.Stop;
                            }
                        }
                    }
                    return Link.Continue;
                });
            return added > 0;
        }

        public static Link[] GetLeftElements(Link startLink, Link leftLink)
        {
            var result = new Link[4];
            TryStepLeft(startLink, leftLink, result, 0);
            startLink.WalkThroughReferersAsSource(couple =>
                {
                    if (couple.Linker == Net.And)
                    {
                        if (TryStepLeft(couple, leftLink, result, 2))
                        {
                            return Link.Stop;
                        }
                    }
                    return Link.Continue;
                });
            return result;
        }

        public static bool TryStepLeft(Link startLink, Link leftLink, Link[] result, int offset)
        {
            var added = 0;
            startLink.WalkThroughReferersAsTarget(couple =>
                {
                    if (couple.Linker == Net.And)
                    {
                        var coupleSource = couple.Source;
                        if (coupleSource == leftLink)
                        {
                            result[offset] = couple;
                            if (++added == 2)
                            {
                                return Link.Stop;
                            }
                        }
                        else if (coupleSource.Linker == Net.And && coupleSource.Target ==
                        ↪  leftLink)
                        {
                            result[offset + 1] = couple;
                            if (++added == 2)
                            {
                                return Link.Stop;
                            }
                        }
                    }
                    return Link.Continue;
                });
            return added > 0;
        }
    }
}
```

1.12 ./csharp/Platform.Data.Triplets.Tests/LinkTests.cs

```
using System.IO;
using Xunit;
using Platform.Random;
using Platform.Ranges;

namespace Platform.Data.Triplets.Tests
{
    public static class LinkTests
    {
        public static object Lock = new object(); //-V3090

        private static ulong _thingVisitorCounter;
        private static ulong _isAVisitorCounter;
        private static ulong _linkVisitorCounter;

```

```
16          static void ThingVisitor(Link linkIndex)
17          {
18              _thingVisitorCounter += linkIndex;
19          }
20
21          static void IsAVisitor(Link linkIndex)
22          {
23              _isAVisitorCounter += linkIndex;
24          }
25
26          static void LinkVisitor(Link linkIndex)
27          {
28              _linkVisitorCounter += linkIndex;
29          }
30
31          [Fact]
32          public static void CreateDeleteLinkTest()
33          {
34              lock (Lock)
35              {
36                  string filename = "db.links";
37
38                  File.Delete(filename);
39
40                  Link.StartMemoryManager(filename);
41
42                  Link link1 = Link.Create(Link.Itself, Link.Itself, Link.Itself);
43
44                  Link.Delete(link1);
45
46                  Link.StopMemoryManager();
47
48                  File.Delete(filename);
49              }
50          }
51
52          [Fact]
53          public static void DeepCreateUpdateDeleteLinkTest()
54          {
55              lock (Lock)
56              {
57                  string filename = "db.links";
58
59                  File.Delete(filename);
60
61                  Link.StartMemoryManager(filename);
62
63                  Link isA = Link.Create(Link.Itself, Link.Itself, Link.Itself);
64                  Link isNotA = Link.Create(Link.Itself, Link.Itself, isA);
65                  Link link = Link.Create(Link.Itself, isA, Link.Itself);
66                  Link thing = Link.Create(Link.Itself, isNotA, link);
67
68                  //Assert::IsTrue(GetLinksCount() == 4);
69
70                  Assert.Equal(isA, isA.Target);
71
72                  isA = Link.Update(isA, isA, isA, link); // Произведено замыкание
73
74                  Assert.Equal(link, isA.Target);
75
76                  Link.Delete(isA); // Одна эта операция удалит все 4 связи
77
78                  //Assert::IsTrue(GetLinksCount() == 0);
79
80                  Link.StopMemoryManager();
81
82                  File.Delete(filename);
83              }
84          }
85
86          [Fact]
87          public static void LinkReferersWalkTest()
88          {
89              lock (Lock)
90              {
91                  string filename = "db.links";
92
93                  File.Delete(filename);
94
95                  Link.StartMemoryManager(filename);
```

```
            Link isA = Link.Create(Link.Itself, Link.Itself, Link.Itself);
            Link isNotA = Link.Create(Link.Itself, Link.Itself, isA);
            Link link = Link.Create(Link.Itself, isA, Link.Itself);
            Link thing = Link.Create(Link.Itself, isNotA, link);
            isA = Link.Update(isA, isA, isA, link);

            Assert.Equal(1, thing.ReferersBySourceCount);
            Assert.Equal(2, isA.ReferersByLinkerCount);
            Assert.Equal(3, link.ReferersByTargetCount);

            _thingVisitorCounter = 0;
            _isAVisitorCounter = 0;
            _linkVisitorCounter = 0;

            thing.WalkThroughReferersAsSource(ThingVisitor);
            isA.WalkThroughReferersAsLinker(IsAVisitor);
            link.WalkThroughReferersAsTarget(LinkVisitor);

            Assert.Equal(4UL, _thingVisitorCounter);
            Assert.Equal(1UL + 3UL, _isAVisitorCounter);
            Assert.Equal(1UL + 3UL + 4UL, _linkVisitorCounter);

            Link.StopMemoryManager();

            File.Delete(filename);
        }
    }

    [Fact]
    public static void MultipleRandomCreationsAndDeletionsTest()
    {
        lock (Lock)
        {
            string filename = "db.links";

            File.Delete(filename);

            Link.StartMemoryManager(filename);

            TestMultipleRandomCreationsAndDeletions(2000);

            Link.StopMemoryManager();

            File.Delete(filename);
        }
    }

    private static void TestMultipleRandomCreationsAndDeletions(int
    ↪ maximumOperationsPerCycle)
    {
        var and = Link.Create(Link.Itself, Link.Itself, Link.Itself);
        //var comparer = Comparer<TLink>.Default;
        for (var N = 1; N < maximumOperationsPerCycle; N++)
        {
            var random = new System.Random(N);
            var linksCount = 1;
            for (var i = 0; i < N; i++)
            {
                var createPoint = random.NextBoolean();
                if (linksCount > 2 && createPoint)
                {
                    var linksAddressRange = new Range<ulong>(1, (ulong)linksCount);
                    Link source = random.NextUInt64(linksAddressRange);
                    Link target = random.NextUInt64(linksAddressRange); //-V3086
                    var resultLink = Link.Create(source, and, target);
                    if (resultLink > linksCount)
                    {
                        linksCount++;
                    }
                }
                else
                {
                    Link.Create(Link.Itself, Link.Itself, Link.Itself);
                    linksCount++;
                }
            }
            for (var i = 0; i < N; i++)
            {
```

```
                    Link link = i + 2;
                    if (link.Linker != null)
                    {
                        Link.Delete(link);
                        linksCount--;
                    }
                }
            }
        }
    }
}
```

1.13 ./csharp/Platform.Data.Triplets.Tests/PersistentMemoryManagerTests.cs

```
using System.IO;
using Xunit;

namespace Platform.Data.Triplets.Tests
{
    public static class PersistentMemoryManagerTests
    {
        [Fact]
        public static void FileMappingTest()
        {
            lock (LinkTests.Lock)
            {
                string filename = "db.links";

                File.Delete(filename);

                Link.StartMemoryManager(filename);

                Link.StopMemoryManager();

                File.Delete(filename);
            }
        }

        [Fact]
        public static void AllocateAndFreeLinkTest()
        {
            lock (LinkTests.Lock)
            {
                string filename = "db.links";

                File.Delete(filename);

                Link.StartMemoryManager(filename);

                Link link = Link.Create(Link.Itself, Link.Itself, Link.Itself);

                Link.Delete(link);

                Link.StopMemoryManager();

                File.Delete(filename);
            }
        }

        [Fact]
        public static void AttachToUnusedLinkTest()
        {
            lock (LinkTests.Lock)
            {
                string filename = "db.links";

                File.Delete(filename);

                Link.StartMemoryManager(filename);

                Link link1 = Link.Create(Link.Itself, Link.Itself, Link.Itself);
                Link link2 = Link.Create(Link.Itself, Link.Itself, Link.Itself);

                Link.Delete(link1); // Creates "hole" and forces "Attach" to be executed

                Link.StopMemoryManager();

                File.Delete(filename);
            }
        }

```

```
68          [Fact]
69          public static void DetachToUnusedLinkTest()
70          {
71              lock (LinkTests.Lock)
72              {
73                  string filename = "db.links";
74
75                  File.Delete(filename);
76
77                  Link.StartMemoryManager(filename);
78
79                  Link link1 = Link.Create(Link.Itself, Link.Itself, Link.Itself);
80                  Link link2 = Link.Create(Link.Itself, Link.Itself, Link.Itself);
81
82                  Link.Delete(link1); // Creates "hole" and forces "Attach" to be executed
83                  Link.Delete(link2); // Removes both links, all "Attached" links forced to be
                        ↪  "Detached" here
84
85                  Link.StopMemoryManager();
86
87                  File.Delete(filename);
88              }
89          }
90
91          [Fact]
92          public static void GetSetMappedLinkTest()
93          {
94              lock (LinkTests.Lock)
95              {
96                  string filename = "db.links";
97
98                  File.Delete(filename);
99
100                 Link.StartMemoryManager(filename);
101
102                 var mapped = Link.GetMappedOrDefault(0);
103
104                 var mappingSet = Link.TrySetMapped(mapped, 0);
105
106                 Assert.True(mappingSet);
107
108                 Link.StopMemoryManager();
109
110                 File.Delete(filename);
111             }
112         }
113     }
114 }
```

Index