

Теория связей 0.0.2

Инженерия данных*Открытый исходный код*Математика*Ненормальное программирование*

Программирование*

Перевод

Автор оригинала: Vasily Solopov, Roman Vertushkin, Ivan Glazunov, Konstantin Diachenko

1 апреля прошлого года, как вы, возможно, догадались, мы шутили. Пора это исправить, и теперь всё 100% серьёзно.

TL;DR (слишком длинно; не читал)

В этой статье содержится много букв, но ее можно представить с помощью всего 4 символов из [теории множеств](#):

$$L \rightarrow L^2$$

Все остальное вытекает из них.

Обзор

Эта статья в первую очередь предназначена для программистов и математиков, однако мы постарались сделать её доступной для всех, кому интересны представленные в ней идеи. Мы считаем, что обсуждаемые здесь концепции могут послужить источником вдохновения для широкого круга научных дисциплин.

Наша цель состояла в том, чтобы создать самостоятельный текст, который проведет вас по каждой теме в чётком и логичном порядке. В статье вы найдёте ссылки на [Википедию](#) для тех, кто хочет более подробно изучить конкретные термины или концепции, но это совершенно необязательно. Текст должен быть понятным при прочтении от начала до конца.

Каждый символ и формула поясняются отдельно, с краткими определениями, где это необходимо. Мы также добавили изображения для иллюстрации ключевых идей. Если вам что-то непонятно, пожалуйста, сообщите нам, и мы сможем это исправить.

Сравнение теорий

Для быстрого погружения начнем со сравнения математических основ двух самых популярных [моделей данных](#) с [ассоциативной моделью данных](#).

В ходе нашего исследования мы обнаружили, что традиционные теории иногда были слишком сложными или избыточными, а в других случаях они налагали слишком много искусственных ограничений.

Этот общий недостаток гибкости, адаптивности и универсальности побудил нас к поиску более простой, но всеобъемлющей информационной теории и модели хранения данных, которые будущий искусственный интеллект мог бы легко понять и эффективно использовать. В процессе мы черпали вдохновение из работы нашей собственной ассоциативной памяти и ассоциативных мыслительных процессов.

Реляционная алгебра

Реляционная алгебра и реляционная модель основаны на концепциях **отношений** и **n-кортежей**.

Отношение определяется как набор из **n-**кортежей :

$$R \subseteq S_1 \times S_2 \times \cdots \times S_n. [1]$$

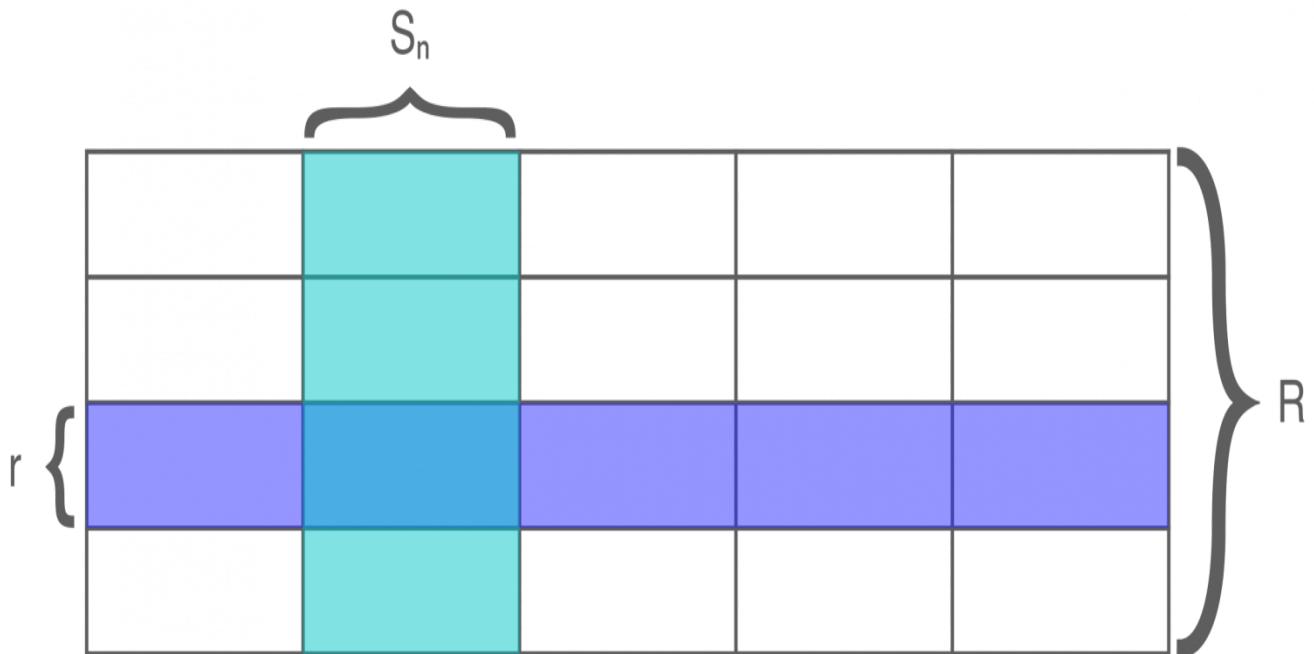


Рисунок 1. Таблица описывается отношением , R которое представлено как набор строк , r принадлежащих декартову произведению $S_1 \times S_2 \times \cdots \times S_n$.

Где:

- Символ обозначает **отношение** (**таблицу**) ; R
- Символ указывает на то , \subseteq что левая часть выражения **является** подмножеством **правой** части;
- Символ обозначает декартово **произведение** двух **множеств** ; \times
- Выражение представляет **домен** , т. е. **набор** всех возможных **значений** , которые может содержать каждая **ячейка** в **столбце** . S_n

Строки или элементы отношения представлены в виде **n -кортежей** . R

Данные в **реляционной модели** группируются в **отношения** . Используя **n-кортежи** в этой **модели** , можно точно представить любую мыслимую **строку данных** , если только мы на самом деле когда-либо использовали **n-кортежи** для этого. И нужны ли вообще **n-кортежи** ? Например, каждый **n-кортеж** можно представить в виде **вложенных упорядоченных пар** , что говорит о том, что одних **упорядоченных пар** может быть достаточно для представления любых данных . Более того, значения **столбцов** в **таблицах** редко представляются в виде **n-кортежей** (хотя, например, **число** можно **разложить** на **n-кортеж** битов). В некоторых **базах данных SQL** даже запрещено использовать более **столбцов** в **таблице** (и, как следствие, в ее соответствующем **n-кортеже**). Таким образом, фактическое значение обычно меньше,

чем . Следовательно, в этих случаях нет настоящих n -кортежей — даже в современных реляционных базах данных . **32 п 32**

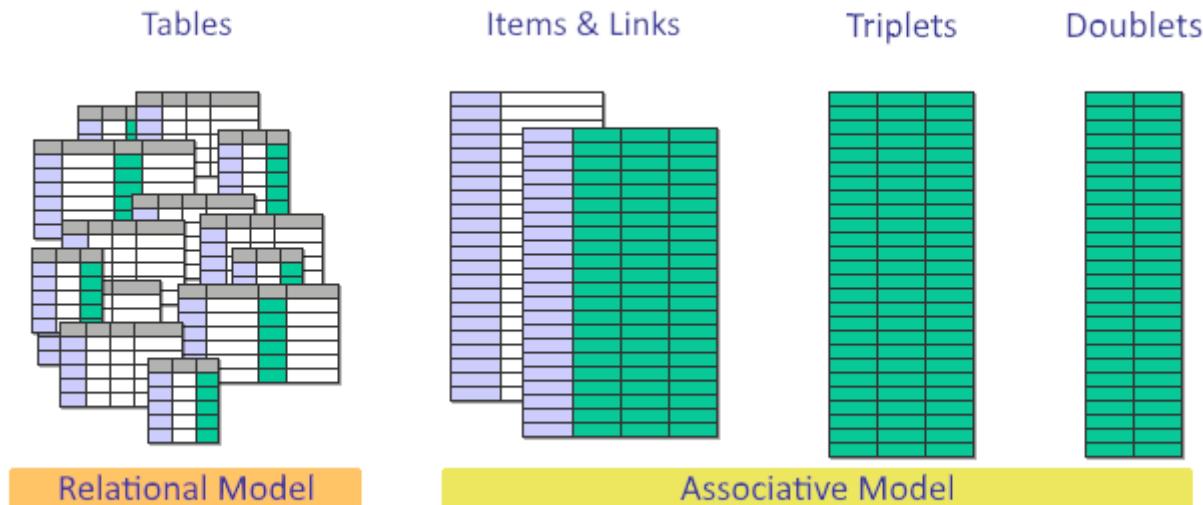


Рисунок 2. Сравнение реляционной модели и ассоциативной модели данных (исходная модель , предложенная Саймоном Уильямсом, была нами упрощена дважды) [3] . Другими словами, представление всех данных в реляционной модели требует множества таблиц — по одной для каждого типа данных — тогда как в ассоциативной модели изначально оказалось достаточно двух таблиц (items и links), а в конечном итоге — одной таблицы (links) триплетных или дублетных связей.

Направленный граф

Ориентированные графы — и графы вообще — основаны на концепциях вершин и ребер (2-кортежей).

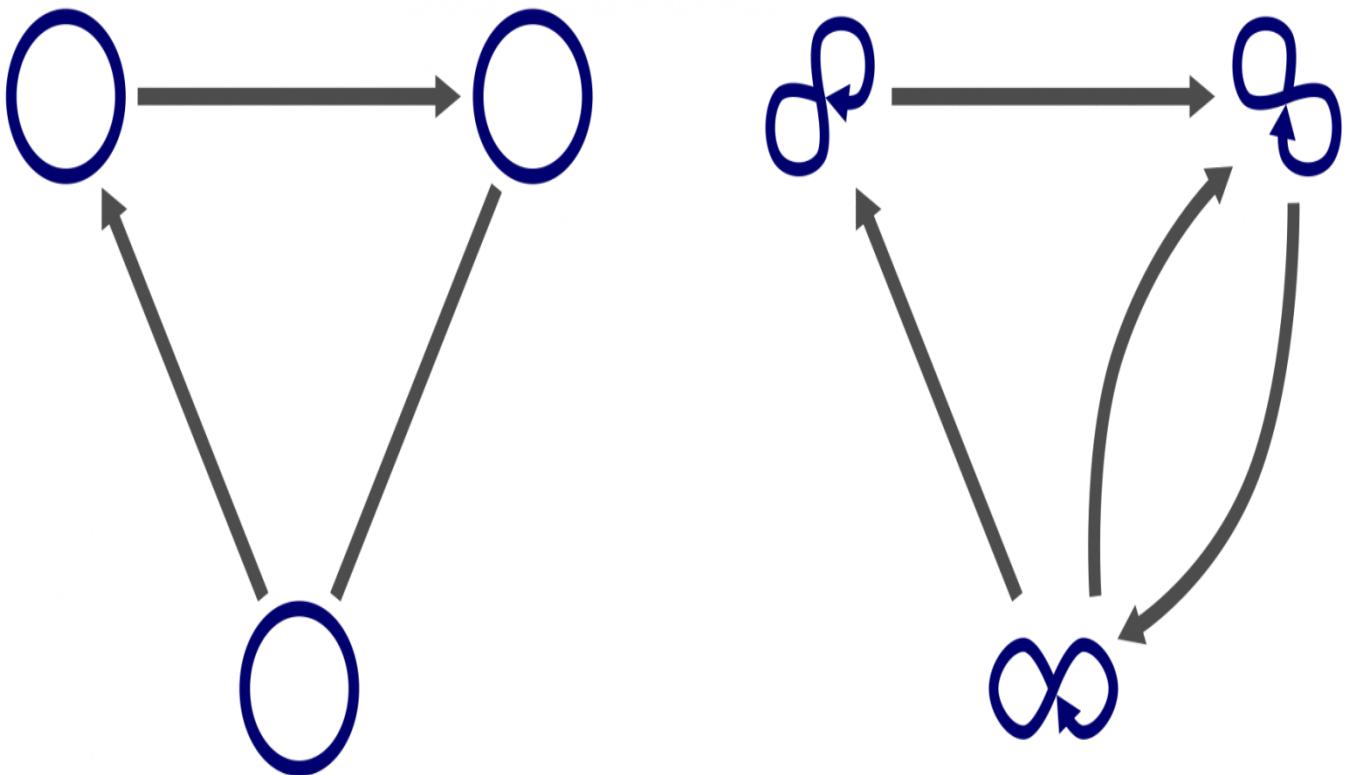
Ориентированный граф G определяется следующим образом:

$$G = (V, E), \quad E \subseteq V \times V. \quad [2]$$

Где:

- V — это множество , элементы которого называются вершинами , узлами или точками ;
- E представляет собой набор упорядоченных пар (кортежей по 2) вершин , называемых дугами, направленными ребрами (иногда просто ребрами), стрелками или направленными отрезками прямых .

В модели направленного графа данные представлены двумя отдельными множествами : узлами и рёбрами . Эта модель может быть использована для представления практически всех структур данных , за исключением, пожалуй, последовательностей (n -кортежей). Иногда для представления последовательностей используются цепочки вершин. Хотя этот метод работает, он неизменно приводит к дублированию данных, а дедупликация в таких случаях либо сложна, либо невозможна. Более того, последовательности в графах можно представить путём разложения последовательности на вложенные множества , но, по нашему мнению, это непрактичный подход. Похоже, мы не одиноки в этом убеждении, что может объяснять, почему нам не встречались примеры использования такого метода другими.



Graph theory

Links theory

Рисунок 3. Сравнение теории графов и теории связей. Вершина эквивалентна [самореферентной связи](#) — связи, которая начинается и заканчивается в самой себе. Направленное ребро представляется в виде направленной пары связей, а ненаправленное ребро — в виде пары направленных связей в противоположных направлениях. Другими словами, в то время как теория графов требует двух типов сущностей — вершин и рёбер, — в теории связей необходимы только связи (которые наиболее близки к рёбрам).

Теория связей

Теория связей основана на концепции связи.

В проекции теории связей на теорию множеств [связь](#) определяется как [n-кортеж](#) ссылок на связи, имеющий собственную ссылку, которую другие связи могут использовать для ссылок на неё.

Стоит отметить, что отдельное понятие ссылки здесь необходимо исключительно потому, что в теории множеств отсутствуют циклические определения. Фактически, теория связей может описывать себя без необходимости в отдельном термине для ссылки — другими словами, ссылка — это просто частный случай связи.

Дуплеты

Дублетная ссылка представлена дуплетом (кортежем из двух или [упорядоченной парой](#)) ссылок на ссылки. Дублетная ссылка также имеет свою собственную ссылку.

$L = \{ 1, 2 \}$

$L \times L = \{$

```

(1, 1),
(1, 2),
(2, 1),
(2, 2),
}

```

Где :

- L — это набор ссылок (от английского слова «Links» как в «References»).

В этом примере набор L содержит только 2 ссылки на ссылки, а именно 1 и 2. Другими словами, в сети ссылок, построенной на таком наборе ссылок, могут быть только 2 ссылки.

Для получения всех возможных значений связи используется [декартово произведение](#) самой себя, т. е . . $L \times L$

	1	2
1	(1, 1)	(1, 2)
2	(2, 1)	(2, 2)

Рисунок 4. Матрица, представляющая декартово произведение множества {1, 2} на само себя. Здесь мы видим, что ссылки с двумя ссылками на ссылки могут иметь только 4 возможных значения.

from	to
1	1
1	2
2	1
2	2

Рисунок 5. Таблица строк, содержащая все возможные варианты значений связей для сети с двумя связями; эти варианты получены с помощью декартова произведения {1, 2} самого на себя.

Сеть **дублетных связей** определяется как:

$$\lambda : L \rightarrow L \times L$$

Где:

- \rightarrow обозначает [отображение \(функцию\)](#) ;
- λ представляет собой функцию, определяющую сеть дуплетных связей;
- L обозначает набор ссылок на ссылки.

Пример :

$$\begin{aligned}
 1 &\rightarrow (1, 1) \\
 2 &\rightarrow (2, 2) \\
 3 &\rightarrow (1, 2)
 \end{aligned}$$

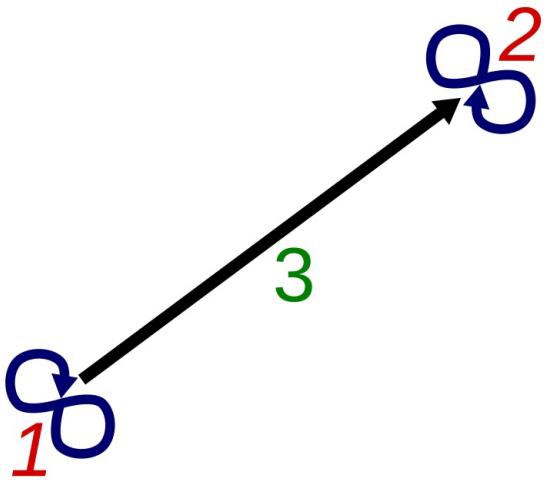


Рисунок 6. Сеть из трёх звеньев. Изображение сети дуплетных звеньев напоминает граф, но мы называем эту визуализацию сетью звеньев. Первое и второе звенья имеют схожую структуру: оба начинаются и заканчиваются в себе. В результате вместо традиционного изображения вершины как точки в теории графов мы получаем графическое представление замкнутой самореферентной стрелки, напоминающей символ бесконечности.

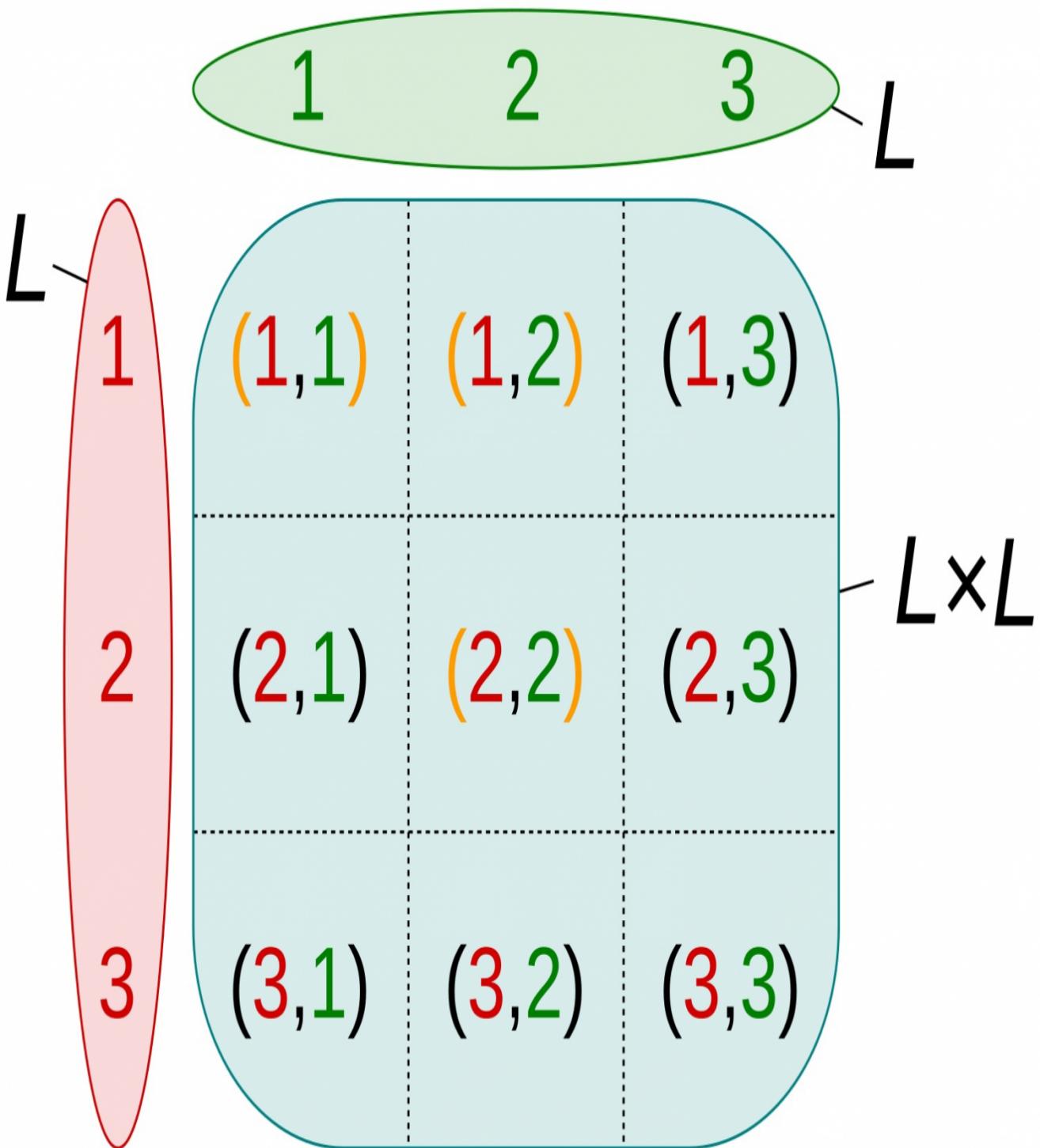


Рисунок 7. Графическое представление декартова произведения в виде матрицы, отображающей все возможные значения связей. Связи, определяющие конкретную сеть , выделены оранжевым цветом. Другими словами, из 9 возможных значений связей выбираются только 3, соответствующие размеру множества L .

Сеть дублетных связей может представлять любую структуру данных.

Например, дублетные связи могут:

- Связать объект с его свойствами;
- Соединить две связи вместе, что не допускается определением теории графов;
- Представить любую последовательность (кортеж из n элементов) в виде дерева, построенного из вложенных упорядоченных пар;

- Опишите предложение на естественном языке, например, с помощью лингвистической модели «[подлежащее-предикат](#)» .

Благодаря этому и другим фактам мы считаем, что дублетные связи могут представлять любую мыслимую структуру данных.

Тройняшки

Триплет-ссылка представлена тройкой (кортежем из 3) ссылок на ссылки.

```
L = { 1 , 2 }
```

```
L × L = {  
  (1, 1),  
  (1, 2),  
  (2, 1),  
  (2, 2),  
}
```

```
L × L × L = {  
  (1, 1, 1),  
  (1, 1, 2),  
  (1, 2, 1),  
  (1, 2, 2),  
  (2, 1, 1),  
  (2, 1, 2),  
  (2, 2, 1),  
  (2, 2, 2),  
}
```

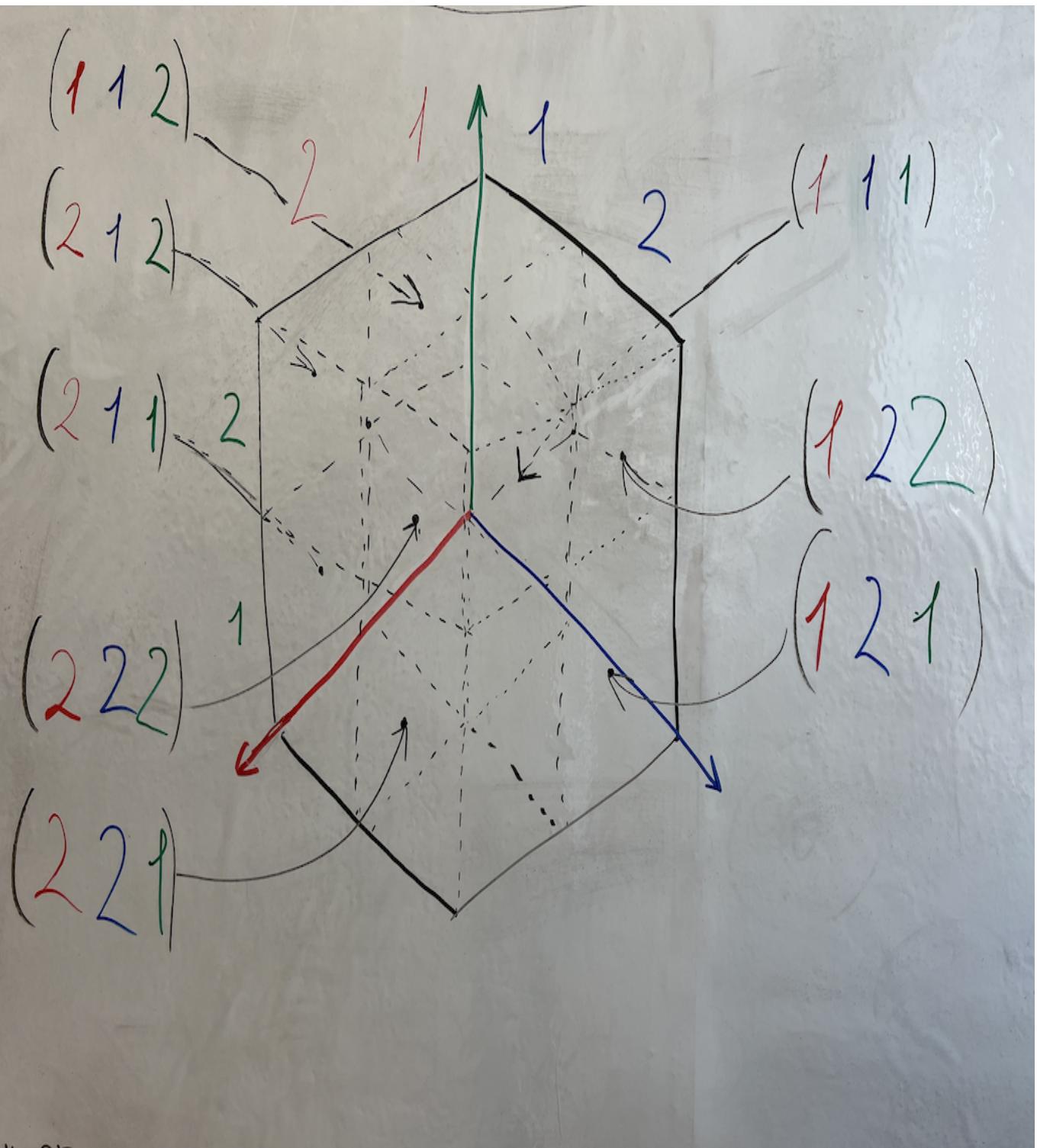


Рисунок 8. Трёхмерная кубическая матрица, представляющая все возможные значения триплетной связи. Такой куб получается рекурсивным декартовым произведением множества $\{1, 2\}$ на себя, то есть $\{1, 2\} \times \{1, 2\} \times \{1, 2\}$.

from	type	to
1	1	1
1	1	2
1	2	1
1	2	2
2	1	1
2	1	2
2	2	1
2	2	2

Рисунок 9. Таблица всех возможных вариантов значений триплетных связей, которые можно получить, рекурсивно умножая множество { 1, 2 } само на себя, то есть $\{ 1, 2 \} \times \{ 1, 2 \} \times \{ 1, 2 \}$. **Примечание:** Первая ссылка может быть интерпретирована как начало, вторая — как тип, а третья — как конец; пользователь определяет, как интерпретировать компоненты вектора ссылки, в соответствии с поставленной задачей.

Сеть **триплетных связей** определяется как:

$$\lambda : \mathbf{L} \rightarrow \mathbf{L} \times \mathbf{L} \times \mathbf{L}$$

Где:

- λ обозначает функцию, определяющую сеть триплетных связей;
- \mathbf{L} обозначает набор ссылок на ссылки.

Пример функции, определяющей конкретную сеть триплетных связей:

$$\begin{aligned} 1 &\rightarrow (1, 1, 1) \\ 2 &\rightarrow (2, 2, 2) \\ 3 &\rightarrow (3, 3, 3) \\ 4 &\rightarrow (1, 2, 3) \end{aligned}$$

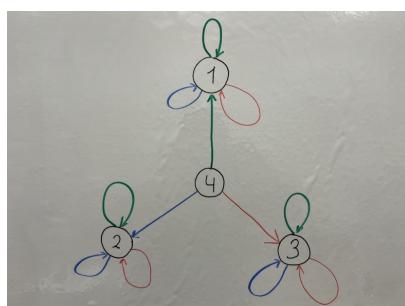


Рисунок 10. Ассоциативная триплетная сеть, представленная в виде цветного ориентированного графа. В этой ассоциативной сети имеется 4 триплета-связи, соответствующих функции, определённой выше. Узлы соответствуют связям, а цвета рёбер соответствуют ссылкам на связи, как показано на рисунке 9 (красный — от, синий — тип, зелёный — к).

Триплетные ссылки могут выполнять те же функции, что и дублетные ссылки. Поскольку триплетные ссылки включают дополнительную ссылку, этот дополнительный элемент

может, например, использоваться для указания типа ссылки.

Например, триплетные связи могут:

- Связать объект, его свойство и его значение;
- Связать две ссылки вместе, используя определенное отношение;
- Опишите предложение на естественном языке, например, используя модель «[подлежащее-глагол-объект](#)» .

Последовательности

Последовательность ссылок, также известная как [n-кортеж](#), является общим случаем.

В общем случае сеть связей определяется как:

$$\lambda : \mathbf{L} \rightarrow \underbrace{\mathbf{L} \times \mathbf{L} \times \dots \times \mathbf{L}}_n$$

Где:

- Символ λ обозначает функцию, определяющую сеть связей;
- Символ \mathbf{L} обозначает набор ссылок.

Пример:

$$\begin{aligned} 1 &\rightarrow (1) \\ 2 &\rightarrow (2, 2) \\ 3 &\rightarrow (3, 3, 3) \\ 4 &\rightarrow (\mathbf{1, 2, 3, 2, 1}) \end{aligned}$$

В этом примере в качестве значений связей используются n-кортежи переменной длины.

Последовательности (векторы), по сути, эквивалентны по выразительной силе реляционной модели — факт, который ещё предстоит доказать в рамках развивающейся теории. Однако, заметив, что дублетных и триплетных связей достаточно для представления последовательностей любого размера, мы выдвинули гипотезу, что нет необходимости использовать последовательности напрямую, поскольку они могут быть представлены двойными связями.

Сравнительный обзор

Реляционная [модель данных](#) может представлять все — даже [ассоциативную модель](#), но для этого необходимо ввести [хороший порядок](#), который обычно представлен в форме отдельного столбца ID. Поскольку реляционная модель данных основана на понятии [набора](#), а не [последовательности](#). Напротив, графовая модель отлично подходит для представления отношений, но менее эффективна для представления уникальных дедуплицированных последовательностей.

Хотя сама реляционная модель не требует разделения данных по нескольким таблицам, традиционные реализации обычно используют этот подход с фиксированными схемами, что приводит к фрагментации связанных данных и усложняет реконструкцию неотъемлемых отношений. Напротив, ассоциативная модель использует унифицированное хранилище ссылок, которое обеспечивает максимально возможную

степень нормализации. Такая конструкция упрощает однозначное сопоставление бизнес-доменов, тем самым облегчая быстрое изменение требований [4].

Ассоциативная модель позволяет легко представить n -кортежи неограниченной длины, используя кортежи с $n \geq 2$. Она столь же эффективна в представлении ассоциаций, как теория графов, и столь же мощна, как реляционная модель, поскольку способна полностью представить любую таблицу SQL. Кроме того, ассоциативная модель может представлять строгие последовательности, позволяя инкапсулировать любую последовательность в уникальную единую ссылку, что полезно для дедупликации.

В реляционной модели для имитации поведения ассоциативной модели требуется только одно отношение, и обычно требуется не более 2–3 столбцов, помимо явного идентификатора или встроенного идентификатора строки. Сам идентификатор необходим в реляционной модели, поскольку она построена на концепции множества, в отличие от теории связей, которая строится на концепции последовательности, поэтому явный идентификатор не требуется (и может быть добавлен при необходимости).

По определению, графовая модель не может напрямую создавать ребро между рёбрами. Таким образом, потребуется либо переопределение, либо расширение с однозначным методом хранения уникальных дедуплицированных последовательностей. Хотя последовательности могут храниться в виде **вложенных множеств** внутри графовой модели, такой подход не пользуется популярностью. Хотя графовая модель наиболее близка к модели дублетных связей, она всё же отличается по определению.

Использование ассоциативной модели означает, что больше нет необходимости выбирать между базами данных SQL и NoSQL; вместо этого ассоциативное хранилище данных может представить все максимально простым способом, при этом данные всегда будут храниться в форме, наиболее близкой к оригиналу (обычно это нормализованная форма, но при необходимости возможна и денормализация).

Математическое введение в теорию связей

Введение

Теперь, когда мы вкратце рассказали об истоках нашей работы, пришло время углубиться в теорию.

Теория связей разрабатывается как более фундаментальная концепция по сравнению с теорией множеств или теорией типов, а также как замена реляционной алгебры и теории графов в качестве объединяющей теории. В то время как теория типов строится на базовых понятиях «тип» и «терм», а теория множеств — на «множестве» и «элементе», теория связей сводит всё к единому понятию «связь».

В этом разделе мы объясним основные понятия и терминологию, используемые в теории связей.

Затем мы дадим определения теории связей в рамках теории множеств, а затем перенесём эти определения в теорию типов с помощью интерактивного доказательчика теорем Coq.

Наконец, мы подведем итоги наших выводов и наметим направления дальнейших исследований и развития теории связей.

Теория связей

В основе теории связей лежит единая концепция связи. Дополнительное понятие ссылки на связь вводится только для теорий, не поддерживающих циклические определения, таких как теория множеств и теория типов.

Связь

Связь обладает асимметричной рекурсивной (фрактальной) структурой, которую можно просто выразить так: связь **связывает** связи (как в выражении «связь **соединяет** связи»). Термин «асимметричная» означает, что каждая связь имеет направление — от её источника (начала) к её назначению (концу).

Определения теории связей в рамках теории множеств

Ссылка **на вектор** — это уникальный идентификатор или порядковый номер, который связан с определенным вектором, представляющим собой последовательность ссылок на другие векторы.

Набор ссылок на векторы:

$$L \subseteq \mathbb{N}_0$$

Вектор **ссылок** — это вектор, состоящий из нуля или более ссылок на векторы, где количество ссылок соответствует количеству элементов в векторе.

Множество всех векторов ссылок длины $n \in \mathbb{N}_0$:

$$V_n = L^n$$

Декартова степень L^n всегда даёт вектор длины n , поскольку все его компоненты имеют один и тот же тип L .

Другими словами, L^n представляет собой множество всех возможных векторов из n элементов (по сути, кортежей из n элементов), в которых каждый элемент принадлежит множеству L .

Ассоциация — это упорядоченная пара, состоящая из ссылки на вектор и вектора ссылок. Эта структура служит для сопоставления ссылок и векторов.

$$A = L \times V_n$$

Ассоциативная **сеть** векторов длины n (или n -мерная ассоциативная сеть) определяется семейством функций $\{anetv^n\}$, где каждая функция $anetv^n : L \rightarrow V_n$ сопоставляет ссылку $l \in L$ с вектором ссылок длины n , принадлежащим V_n , тем самым идентифицируя точки в n -мерном пространстве.

n В $anetv^n$ означает, что функция возвращает векторы, содержащие n ссылки. Таким образом, каждая n -мерная ассоциативная сеть представляет собой последовательность точек в n -мерном пространстве.

Семейство функций:

$$\cup_f \{anetv^n | n \in \mathbb{N}_0\} \subseteq A$$

Здесь символ объединения \cup обозначает объединение всех функций в семействе $\{anetv^n\}$, а символ \subseteq указывает на то, что эти упорядоченные пары, рассматриваемые как функциональные бинарные отношения, являются подмножеством множества A всех ассоциаций.

Набор дуплетов (упорядоченных пар или двумерных векторов) ссылок:

$$D = L^2$$

Это множество всех дуплетов (L, L) , т.е. вторая декартова степень числа L .

Ассоциативная сеть дуплетов (или двумерная ассоциативная сеть):

$$\text{anetd} : L \rightarrow L^2$$

Таким образом, каждая ассоциативная сеть дуплетов представляет собой последовательность точек в двумерном пространстве.

Пустой вектор (вектор нулевой длины) представлен пустым кортежем, обозначаемым как $()$ или \emptyset .

Ассоциативная сеть вложенных упорядоченных пар:

$$\text{anetl} : L \rightarrow NP, \text{ where } NP = \{(\emptyset, \emptyset) | (l, np), l \in L, np \in NP\}$$

NP — это множество вложенных упорядоченных пар, состоящее из пустых пар и пар, содержащих один или более элементов. Таким образом, вектор длины $n \in \mathbb{N}_0$ можно представить в виде вложенных упорядоченных пар.

Проекция теории связей в теорию типов (Coq) через теорию множеств

О Coq

Coq — это интерактивный доказатель теорем, основанный на теории типов высшего порядка, также известной как исчисление индуктивных построений (CIC). Это мощная среда для формализации сложных математических теорем, проверки корректности доказательств и извлечения исполняемого кода из формально верифицированных спецификаций. Coq широко используется как в академической среде для формализации математики, так и в ИТ-индустрии для верификации программного и аппаратного обеспечения.

Решение использовать Coq для описания теории связей в рамках теории типов было обусловлено необходимостью строгой формализации доказательств и обеспечения логической корректности при разработке теории связей. Coq обеспечивает точное выражение свойств и операций над связями благодаря своей надёжной системе типов и развитым механизмам доказательств.

В преддверии обширной работы, направленной на доказательство эквивалентности реляционной модели и ассоциативной сети дублетов, в этом разделе представлены первые шаги, предпринятые с использованием системы доказательств Coq. На первом этапе наша цель — формализовать структуры ассоциативных сетей, определив базовые типы, функции и структуры в Coq.

Определения ассоциативных сетей

[[Ссылка на исходный код](#)]

```
Require Import PeanoNat.  
Require Import Coq.Init.Nat.  
Require Import Vector.  
Require Import List.  
Require Import Coq.Init.Datatypes.  
Import ListNotations.  
Import VectorNotations.
```

(* Set of vector references: L ⊆ № *)

```

Definition L := nat.

(* Default value for L: zero *)
Definition LDefault : L := 0.

(* Set of vectors of references of length n  $\in \mathbb{N}_0$ :  $V_n \subseteq L^n$  *)
Definition Vn (n : nat) := t L n.

(* Default value for  $V_n$  *)
Definition VnDefault (n : nat) : Vn n := Vector.const LDefault n.

(* Set of all associations:  $A = L \times V_n$  *)
Definition A (n : nat) := prod L (Vn n).

(* Associative network of vectors of length n (or n-dimensional associative
network) from the family of functions  $\{\text{anetv}^n : L \rightarrow V_n\}$  *)
Definition ANetVf (n : nat) := L -> Vn n.

(* Associative network of vectors of length n (or n-dimensional associative
network) as a sequence *)
Definition ANetVl (n : nat) := list (Vn n).

(* Nested ordered pairs *)
Definition NP := list L.

(* Associative network of nested ordered pairs:  $\text{anetl} : L \rightarrow NP$  *)
Definition ANetLf := L -> NP.

(* Associative network of nested ordered pairs as a sequence of nested ordered
pairs *)
Definition ANetLl := list NP.

(* Duplet of references *)
Definition D := prod L L.

(* Default value for D: a pair of two LDefault values, used to denote an empty
duplet *)
Definition DDefault : D := (LDefault, LDefault).

(* Associative network of duplets (or two-dimensional associative network):
 $\text{anetd} : L \rightarrow L^2$  *)
Definition ANetDf := L -> D.

(* Associative network of duplets (or two-dimensional associative network) as a
sequence of duplets *)
Definition ANetDl := list D.

Функции преобразования ассоциативной сети

(* Function to convert  $V_n$  to NP *)
Fixpoint VnToNP {n : nat} (v : Vn n) : NP :=
  match v with
  | Vector.nil _ => List.nil
  | Vector.cons _ h _ t => List.cons h (VnToNP t)
  end.

(* Function to convert ANetVf to ANetLf *)
Definition ANetVfToANetLf {n : nat} (a: ANetVf n) : ANetLf :=
  fun id => VnToNP (a id).

(* Function to convert ANetVl to ANetLl *)
Definition ANetVlToANetLl {n: nat} (net: ANetVl n) : ANetLl :=
  map VnToNP net.

(* Function to convert NP to  $V_n$ , returning an option *)

```

```

Fixpoint NPToVnOption (n: nat) (p: NP) : option (Vn n) :=
match n, p with
| 0, List.nil => Some (Vector.nil nat)
| S n', List.cons f p' =>
  match NPToVnOption n' p' with
  | None => None
  | Some t => Some (Vector.cons nat f n' t)
  end
| _, _ => None
end.

(* Function to convert NP to Vn using VnDefault *)
Definition NPToVn (n: nat) (p: NP) : Vn n :=
match NPToVnOption n p with
| None => VnDefault n
| Some t => t
end.

(* Function to convert ANetLf to ANetVf *)
Definition ANetLfToANetVf { n: nat } (net: ANetLf) : ANetVf n :=
fun id => match NPToVnOption n (net id) with
| Some t => t
| None => VnDefault n
end.

(* Function to convert ANetLl to ANetVl *)
Definition ANetLlToANetVl {n: nat} (net : ANetLl) : ANetVl n :=
map (NPToVn n) net.

(* Function to convert NP to ANetDl with an index offset *)
Fixpoint NPToANetDl_ (offset: nat) (np: NP) : ANetDl :=
match np with
| nil => nil
| cons h nil => cons (h, offset) nil
| cons h t => cons (h, S offset) (NPToANetDl_ (S offset) t)
end.

(* Function to convert NP to ANetDl *)
Definition NPToANetDl (np: NP) : ANetDl := NPToANetDl_ 0 np.

(* Function to append NP to the tail of ANetDl *)
Definition AddNPToANetDl (anet: ANetDl) (np: NP) : ANetDl :=
app anet (NPToANetDl_ (length anet) np).

(* Function that removes the head of anetd and returns the tail starting at
offset *)
Fixpoint ANetDl_behead (anet: ANetDl) (offset : nat) : ANetDl :=
match offset with
| 0 => anet
| S n' =>
  match anet with
  | nil => nil
  | cons h t => ANetDl_behead t n'
  end
end.

(* Function to convert ANetDl to NP with indexing starting at the beginning of
ANetDl from offset *)
Fixpoint ANetDlToNP_ (anet: ANetDl) (offset: nat) (index: nat): NP :=
match anet with
| nil => nil
| cons (x, next_index) tail_anet =>
  if offset =? index then
    cons x (ANetDlToNP_ tail_anet (S offset) next_index)
  else

```

```

ANetDlToNP_ tail_anet (S offset) index
end.

(* Function to read NP from ANetDl by the duplet index *)
Definition ANetDl_readNP (anet: ANetDl) (index: nat) : NP :=
  ANetDlToNP_ anet 0 index.

(* Function to convert ANetDl to NP starting from the head of the anet list *)
Definition ANetDlToNP (anet: ANetDl) : NP := ANetDl_readNP anet 0.

(*
  Now everything is ready for converting the associative network of nested
  ordered pairs anetl : L → NP
  into the associative network of duplets anetd : L → L2.
)

This conversion can be done in different ways: either preserving the original
references to vectors
or with reindexing. Reindexing can be omitted if one writes an additional
function for the duplet associative network
that returns the nested ordered pair by its reference.
*)

(* Function to add ANetLl to ANetDl *)
Fixpoint AddANetLlToANetDl (anetd: ANetDl) (anetl: ANetLl) : ANetDl :=
  match anetl with
  | nil => anetd
  | cons h t => AddANetLlToANetDl (AddNPToANetDl anetd h) t
end.

(* Function to convert ANetLl to ANetDl *)
Definition ANetLlToANetDl (anetl: ANetLl) : ANetDl :=
  match anetl with
  | nil => nil
  | cons h t => AddANetLlToANetDl (NPToANetDl h) t
end.

(* Function to find NP in the tail of ANetDl starting at offset by its ordinal
number.
  Returns the NP offset. *)
Fixpoint ANetDl_offsetNP_ (anet: ANetDl) (offset: nat) (index: nat) : nat :=
  match anet with
  | nil => offset + (length anet)
  | cons (_, next_index) tail_anet =>
    match index with
    | 0 => offset
    | S index' =>
      if offset =? next_index then
        ANetDl_offsetNP_ tail_anet (S offset) index'
      else
        ANetDl_offsetNP_ tail_anet (S offset) index
    end
  end.

(* Function to find NP in ANetDl by its ordinal number.
  Returns the NP offset. *)
Definition ANetDl_offsetNP (anet: ANetDl) (index: nat) : nat :=
  ANetDl_offsetNP_ anet 0 index.

(* Function to convert ANetVl to ANetDl *)
Definition ANetVlToANetDl {n : nat} (anetv: ANetVl n) : ANetDl :=
  ANetLlToANetDl (ANetVlToANetLl anetv).

(*
  Now everything is ready for converting the duplet associative network anetd :
  L → L2
)
```

into the associative network of nested ordered pairs $\text{anet1} : \text{L} \rightarrow \text{NP}$.

We will perform this conversion while preserving the original references to vectors.

Reindexing can be omitted because there is the function `ANetDl_offsetNP` for the duplet associative network

that returns the offset of the nested ordered pair by its reference.
*)

(* Function that removes the first NP from ANetDl and returns the tail *)

Fixpoint `ANetDl_beheadNP` (`anet: ANetDl`) (`offset: nat`) : `ANetDl` :=

```
match anet with
| nil => nil
| cons (_ , next_index) tail_anet =>
  if offset =? next_index then (* end of NP *)
    tail_anet
  else (* NP not ended yet *)
    ANetDl_beheadNP tail_anet (S offset)
end.
```

(* Function to convert NP and ANetDl with an offset into ANetLl *)

Fixpoint `ANetDlToANetLl_` (`anetd: ANetDl`) (`np: NP`) (`offset: nat`) : `ANetLl` :=

```
match anetd with
| nil => nil (* discard NP even if incomplete *)
| cons (x, next_index) tail_anet =>
  if offset =? next_index then (* end of NP, move to the next NP *)
    cons (app np (cons x nil)) (ANetDlToANetLl_ tail_anet nil (S offset))
  else (* NP not finished yet, continue parsing the duplet network *)
    ANetDlToANetLl_ tail_anet (app np (cons x nil)) (S offset)
end.
```

(* Function to convert ANetDl to ANetLl *)

Definition `ANetDlToANetLl` (`anetd: ANetDl`) : `ANetLl` :=

```
ANetDlToANetLl_ anetd nil LDefault.
```

Предикаты эквивалентности для ассоциативных сетей

(* The definition `ANetVf_equiv` introduces a predicate for the equivalence of two associative networks of vectors of length n ,

anet1 and anet2 of type `ANetVf`.

This predicate describes the property of "equivalence" for such networks.

It asserts that anet1 and anet2 are considered "equivalent" if, for every reference id , the vector associated with id in anet1

exactly matches the vector associated with the same id in anet2 .

*)

Definition `ANetVf_equiv` {`n: nat`} (`anet1: ANetVf n`) (`anet2: ANetVf n`) : `Prop` :=
`forall id, anet1 id = anet2 id.`

(* The definition `ANetV1_equiv_V1` introduces a predicate for the equivalence of two associative networks of vectors of length n ,

anet1 and anet2 of type `ANetV1`.

*)

Definition `ANetV1_equiv_V1` {`n: nat`} (`anet1: ANetV1 n`) (`anet2: ANetV1 n`) : `Prop` :=
`anet1 = anet2.`

(* Equivalence predicate for associative networks of duplets `ANetDf` *)

Definition `ANetDf_equiv` (`anet1: ANetDf`) (`anet2: ANetDf`) : `Prop` := `forall id,`
`anet1 id = anet2 id.`

(* Equivalence predicate for associative networks of duplets `ANetDl` *)

Definition `ANetDl_equiv` (`anet1: ANetDl`) (`anet2: ANetDl`) : `Prop` := `anet1 = anet2.`

Леммы об эквивалентности ассоциативных сетей

```
(* Lemma on preservation of vector length in the associative network *)
Lemma Vn_dim_preserved : forall {l: nat} (t: Vn l), List.length (VnToNP t) = l.
Proof.
  intros l t.
  induction t.
  - simpl. reflexivity.
  - simpl. rewrite IHt. reflexivity.
Qed.
```

(* Lemma on the mutual inversion of the functions NPToVnOption and VnToNP

H_inverse proves that every Vn vector can be converted losslessly to an NP using VnToNP and then back to Vn using NPToVnOption.

Formally, forall n: nat, forall t: Vn n, NPToVnOption n (VnToNP t) = Some t states that

for every natural number n and each Vn vector of length n,
we can convert Vn to NP using VnToNP,
then convert the result back to Vn using NPToVnOption n,
and ultimately obtain the same Vn vector we started with.

This property is very important because it guarantees that these two functions

form an inverse pair on the set of convertible vectors Vn and NP.

When you apply both functions to values in this set, you end up with the original value.

This means that no information is lost during the transformations,

so you can freely convert between Vn and NP as required in implementations or proofs.

*)
Lemma H_inverse: forall n: nat, forall t: Vn n, NPToVnOption n (VnToNP t) = Some t.

Proof.

```
intros n.
induction t as [| h n' t' IH].
- simpl. reflexivity.
- simpl. rewrite IH. reflexivity.
```

Qed.

(*

The Wrapping and Recovery Theorem for the Associative Network of Vectors:

Let an associative network of vectors of length n be given, denoted as anetvn : L → Vⁿ.

Define an operation that maps this network to the associative network of nested ordered pairs anetl : L → NP,

where NP = {(\emptyset, \emptyset) | (l, np), l ∈ L, np ∈ NP}.

Then define the inverse mapping from the associative network of nested ordered pairs back to the associative network of vectors of length n.

The theorem states:

For any associative network of vectors of length n, anetvn, applying the transformation to the associative network

of nested ordered pairs and then the inverse transformation back to the associative network of vectors of length n

recovers the original network anetvn.

In other words:

∀ anetvn : L → Vⁿ, inverse(forward(anetvn)) = anetvn.

*)
Theorem anetf_equiv_after_transforms : forall {n: nat} (anet: ANetVf n),
ANetVf_equiv anet (fun id => match NPToVnOption n ((ANetVfToANetLf anet) id)

```

with
| Some t => t
| None   => anet id
end).

```

Proof.

```

intros n net id.
unfold ANetVfToANetLf.
simpl.
rewrite H_inverse.
reflexivity.

```

Qed.

(* Lemma on preservation of the length of NP lists in the duplet associative network *)

```

Lemma NP_dim_preserved : forall (offset: nat) (np: NP),
  length np = length (NPToANetDl_ offset np).

```

Proof.

```

intros offset np.
generalize dependent offset.
induction np as [| n np' IHnp'];
  intros offset.
- simpl. reflexivity.
- destruct np' as [| m np''];
  simpl; simpl in IHnp'.
  + reflexivity.
  + rewrite IHnp' with (offset := S offset). reflexivity.

```

Qed.

Примеры преобразований между ассоциативными сетями

(* Notation for list notation *)

```

Notation "{ }" := (nil) (at level 0).

```

```

Notation "{ x , .. , y }" := (cons x .. (cons y nil) ..) (at level 0).

```

(* Three-dimensional associative network *)

```

Definition complexExampleNet : ANetVf 3 :=

```

```

  fun id => match id with
  | 0 => [0; 0; 0]
  | 1 => [1; 1; 2]
  | 2 => [2; 4; 0]
  | 3 => [3; 0; 5]
  | 4 => [4; 1; 1]
  | S _ => [0; 0; 0]
end.

```

(* Vectors of references *)

```

Definition exampleTuple0 : Vn 0 := [].

```

```

Definition exampleTuple1 : Vn 1 := [0].

```

```

Definition exampleTuple4 : Vn 4 := [3; 2; 1; 0].

```

(* Conversion of vectors of references into nested ordered pairs (lists) *)

```

Definition nestedPair0 := VnToNP exampleTuple0.

```

```

Definition nestedPair1 := VnToNP exampleTuple1.

```

```

Definition nestedPair4 := VnToNP exampleTuple4.

```

```

Compute nestedPair0. (* Expected result: { } *)

```

```

Compute nestedPair1. (* Expected result: {0} *)

```

```

Compute nestedPair4. (* Expected result: {3, 2, 1, 0} *)

```

(* Computing the values of the converted function of the three-dimensional associative network *)

```

Compute (ANetVfToANetLf complexExampleNet) 0. (* Expected result: {0, 0, 0} *)

```

```

Compute (ANetVfToANetLf complexExampleNet) 1. (* Expected result: {1, 1, 2} *)

```

```

Compute (ANetVfToANetLf complexExampleNet) 2. (* Expected result: {2, 4, 0} *)

```

```

Compute (ANetVfToANetLf complexExampleNet) 3. (* Expected result: {3, 0, 5} *)

```

```

Compute (ANetVfToANetLf complexExampleNet) 4. (* Expected result: {4, 1, 1} *)

```

```

Compute (ANetVfToANetLf complexExampleNet) 5. (* Expected result: {0, 0, 0} *)
(* Associative network of nested ordered pairs *)
Definition testPairsNet : ANetLf :=
  fun id => match id with
    | 0 => {5, 0, 8}
    | 1 => {7, 1, 2}
    | 2 => {2, 4, 5}
    | 3 => {3, 1, 5}
    | 4 => {4, 2, 1}
    | S _ => {0, 0, 0}
  end.

(* Converted associative network of nested ordered pairs into a three-
dimensional associative network (dimensions must match) *)
Definition testTuplesNet : ANetVf 3 :=
  ANetLfToANetVf testPairsNet.

(* Computing the values of the converted function of the associative network of
nested ordered pairs *)
Compute testTuplesNet 0. (* Expected result: [5; 0; 8] *)
Compute testTuplesNet 1. (* Expected result: [7; 1; 2] *)
Compute testTuplesNet 2. (* Expected result: [2; 4; 5] *)
Compute testTuplesNet 3. (* Expected result: [3; 1; 5] *)
Compute testTuplesNet 4. (* Expected result: [4; 2; 1] *)
Compute testTuplesNet 5. (* Expected result: [0; 0; 0] *)

(* Conversion of nested ordered pairs into the associative network of duplets *)
Compute NPToANetDl { 121, 21, 1343 }.
(* Should return: {(121, 1), (21, 2), (1343, 2)} *)

(* Adding nested ordered pairs to the associative network of duplets *)
Compute AddNPToANetDl {(121, 1), (21, 2), (1343, 2)} {12, 23, 34}.
(* Expected result: {(121, 1), (21, 2), (1343, 2), (12, 4), (23, 5), (34, 5)} *)

(* Conversion of the associative network of duplets into nested ordered pairs *)
Compute ANetDlToNP {(121, 1), (21, 2), (1343, 2)}.
(* Expected result: {121, 21, 1343} *)

Compute ANetDlToNP {(121, 1), (21, 2), (1343, 2), (12, 4), (23, 5), (34, 5)}.
(* Expected result: {121, 21, 1343} *)

(* Reading nested ordered pairs from the associative network of duplets by the
duplet index (start of the nested ordered pair) *)
Compute ANetDl_readNP {(121, 1), (21, 2), (1343, 2), (12, 4), (23, 5), (34, 5)} 0.
(* Expected result: {121, 21, 1343} *)

Compute ANetDl_readNP {(121, 1), (21, 2), (1343, 2), (12, 4), (23, 5), (34, 5)} 3.
(* Expected result: {12, 23, 34} *)

(* Defining an associative network of nested ordered pairs *)
Definition test_anetl := { {121, 21, 1343}, {12, 23}, {34}, {121, 21, 1343},
{12, 23}, {34} }.

(* Converted associative network of nested ordered pairs into the associative
network of duplets *)
Definition test_anetd := ANetLfToANetDl test_anetl.

(* Computing the converted associative network of nested ordered pairs into the
associative network of duplets *)
Compute test_anetd.
(* Expected result:
{(121, 1), (21, 2), (1343, 2),

```

```

(12, 4), (23, 4),
(34, 5),
(121, 7), (21, 8), (1343, 8),
(12, 10), (23, 10),
(34, 11) } *)

(* Converting the associative network of nested ordered pairs into the
associative network of duplets and back into test_anetl *)
Compute ANetDlToANetLl test_anetd.
(* Expected result:
 { {121, 21, 1343}, {12, 23}, {34}, {121, 21, 1343}, {12, 23}, {34}} *)

(* Computing the offset of nested ordered pairs in the associative network of
duplets by their ordinal number *)
Compute ANetDl_OffsetNP test_anetd 0. (* Expected result: 0 *)
Compute ANetDl_OffsetNP test_anetd 1. (* Expected result: 3 *)
Compute ANetDl_OffsetNP test_anetd 2. (* Expected result: 5 *)
Compute ANetDl_OffsetNP test_anetd 3. (* Expected result: 6 *)
Compute ANetDl_OffsetNP test_anetd 4. (* Expected result: 9 *)
Compute ANetDl_OffsetNP test_anetd 5. (* Expected result: 11 *)
Compute ANetDl_OffsetNP test_anetd 6. (* Expected result: 12 *)
Compute ANetDl_OffsetNP test_anetd 7. (* Expected result: 12 *)

(* Defining a three-dimensional associative network as a sequence of vectors of
length 3 *)
Definition test_anetv : ANetVl 3 :=
{ [0; 0; 0], [1; 1; 2], [2; 4; 0], [3; 0; 5], [4; 1; 1], [0; 0; 0] }.

(* Converted three-dimensional associative network into the associative network
of duplets via the associative network of nested ordered pairs *)
Definition test_anetdl : ANetDl := ANetVlToANetDl test_anetv.

(* Computing the three-dimensional associative network converted into the
associative network of duplets via the associative network of nested ordered
pairs *)
Compute test_anetdl.
(* Expected result:
 { (0, 1), (0, 2), (0, 2),
 (1, 4), (1, 5), (2, 5),
 (2, 7), (4, 8), (0, 8),
 (3, 10), (0, 11), (5, 11),
 (4, 13), (1, 14), (1, 14),
 (0, 16), (0, 17), (0, 17)} *)

(* Converted three-dimensional associative network into the associative network
of duplets via the associative network of nested ordered pairs and then back
into a three-dimensional associative network *)
Definition result_TuplesNet : ANetVl 3 :=
ANetLlToANetVl (ANetDlToANetLl test_anetdl).

(* Final check of the equivalence of associative networks *)
Compute result_TuplesNet.
(* Expected result:
 { [0; 0; 0], [1; 1; 2], [2; 4; 0], [3; 0; 5], [4; 1; 1], [0; 0; 0]} *)

```

Практическая реализация

Существует несколько практических реализаций: [Deep](#), [LinksPlatform](#) и модель отношений .

Глубокий

[Deep](#) — это система, основанная на теории связей. В теории связей связи могут использоваться для представления любых данных или знаний, а также для

программирования. Deep построен на этой философии: в Deep всё является связью. Однако, если разделить эти связи на две категории, то мы имеем сами данные и поведение. Поведение, представленное кодом в Deep, хранится в ассоциативном хранилище в виде связей и для выполнения передаётся в Docker-контейнер соответствующего языка программирования, где он выполняется изолированно и безопасно. Всё взаимодействие между различными частями кода осуществляется через связи в хранилище (базе данных), что делает базу данных универсальным API для работы с данными (в отличие от традиционной практики вызова функций и методов). В настоящее время в качестве ассоциативного хранилища в Deep используется PostgreSQL, который впоследствии будет заменён движком данных на основе дублетов и триплетов от LinksPlatform.

Deep обеспечивает взаимодействие всего программного обеспечения на планете, представляя все его компоненты в виде ссылок. Также возможно хранить любые данные и код вместе, связывая события или действия различных типов ассоциаций с соответствующим кодом, который выполняется для обработки этих событий. Каждый обработчик может извлекать необходимые ссылки из ассоциативного хранилища и вставлять/обновлять/удалять ссылки в нём, что может инициировать дальнейшее каскадное выполнение обработчиков.

Таблица `links` в базе данных PostgreSQL компании Deep содержит записи, которые можно интерпретировать как ссылки. Они содержат столбцы, такие как `id`, `type_id`, `from_id`, и `to_id`. Типы ссылок помогают разработчику ассоциативных пакетов предопределять семантику отношений между различными элементами, обеспечивая однозначное понимание ссылок как пользователями, так и кодом в ассоциативных пакетах. Помимо `links` таблицы, система также включает таблицы с именами `numbers`, `strings`, и `objects` для хранения числовых, строковых и JSON-значений соответственно. Каждая ссылка может быть связана только с одним значением. Это временное решение, которое используется до тех пор, пока Deep не перейдет на использование LinksPlatform в качестве ядра базы данных. После завершения миграции все эти, казалось бы, базовые типы данных будут построены с нуля, используя только ссылки. Это позволит использовать дедупликацию (которая возникает как следствие теории ссылок) и глубоко понимать внутреннюю структуру значений. Также планируется добавить индексацию таких сложных значений, представленных ссылками, для повышения производительности, чтобы сделать ее такой же быстрой или даже быстрее, чем текущая реализация PostgreSQL.

LinksPlatfrom

[LinksPlatfrom](#) — это кроссплатформенный многоязыковой фреймворк, предназначенный для реализации низкоуровневой ассоциативности в виде конструктора СУБД. Например, в настоящее время у нас есть [бенчмарк](#), сравнивающий реализацию дублетов в PostgreSQL с аналогичной реализацией на чистом Rust/C++; ведущая реализация на Rust превосходит PostgreSQL по 1746 ... 15745 времени выполнения операций записи и 100 ... 9694 чтения.

Модель отношений

[Модель отношений](#) представляет собой язык метапрограммирования, основанный на представлении программы в виде трёхмерной ассоциативной сети. Модель отношений соответствует принципам существенно-ориентированного программирования, где сущность используется как единственное фундаментальное понятие, то есть предполагается, что всё есть сущность и нет ничего, кроме сущностей.

В модели отношений сущность, в зависимости от своего внутреннего конститутивного принципа, может быть либо структурой (объектом), либо функцией (методом). В отличие от известной ER-модели, которая использует для представления схемы базы данных два базовых понятия — сущность и отношение, в модели отношений сущность и отношение по сути являются одним и тем же. Такое представление позволяет описывать не только внешние связи сущности, но и её внутреннюю модель — модель отношений.

Сущность по своему внутреннему принципу триедина (троична, состоит из трех элементов), поскольку она представляет собой синтез трех аспектов (качеств) других сущностей (или самой себя).

jsonRVM — это многопоточная виртуальная машина для выполнения JSON-проекции модели отношений. Модель отношений, представленная в виде JSON, позволяет писать программы непосредственно в JSON. Это представление представляет собой гибрид сегментов данных и кода и позволяет легко десериализовать/выполнять/сериализовать проекцию модели отношений, а также использовать JSON-редакторы для программирования. В процессе выполнения модели отношений метапрограмма может не только обрабатывать данные, но и генерировать многопоточные программы и метапрограммы, выполняя их немедленно или экспортируя в JSON.

Заключение

В этой статье мы рассмотрели математические основы реляционной алгебры и теории графов, а также представили определения теории связей в терминах теории множеств и её проекции на теорию типов. Мы также определили набор функций и лемм, необходимых для доказательства возможности эквивалентного преобразования любого вектора/последовательности во вложенные дублетные связи и обратно. Это означает, что для представления любого возможного типа информации достаточно одной формулы:

$$L \rightarrow L^2$$

Таким образом, это формирует основу для проверки гипотезы о том, что любая другая структура данных может быть представлена дублетными связями. Другими словами, дублетных связей достаточно для представления любых таблиц, графиков, строк, массивов, списков, чисел, звука, изображений, видео и многое другого.

Ещё одним следствием этого доказательства является то, что мы можем представить ленту машины Тьюринга, используя только дублетные связи. Это означает, что связи могут быть столь же мощными по объёму памяти, как машина Тьюринга. То есть мы можем использовать связи во всех случаях, где используется машина Тьюринга. Но нет необходимости пытаться уместить данные в нули и единицы, не нужно ломать над этим голову. Потому что в связях «алфавит», по сути, неограничен, и вы можете добавлять любое количество связей, придавать им любое необходимое значение и связывать или соединять их друг с другом любым нужным вам способом. Обычно это означает, что ваше понятие, объект или вещь из реального или абстрактного мира будут отражены в связях 1 к 1 или максимально приближены к оригиналу, что иногда невозможно с помощью традиционных методов.

Мы продолжаем добиваться прогресса в синхронизации смыслов между нашими тремя проектами и между людьми в нашем ассоциативном сообществе. Эти проекты призваны привнести ассоциативность в мир и сделать её полезной для человечества. Эта статья — очередная итерация нашей дискуссии, позволяющая нам прийти к единому значению слов и терминов в рамках общей ассоциативной теории. Мы считаем, что эта теория может стать метаязыком, на котором люди и машины уже общаются.

С каждым дальнейшим уточнением мы будем на шаг ближе к тому, чтобы говорить на общем языке и сделать эту идею более понятной для всех. Эта теория также будет полезна для различных оптимизаций в разрабатываемых ассоциативных реализациях, а в будущем — для проектирования ассоциативных микросхем (или сопроцессоров) для ускорения операций с данными, представленными в виде связей.

Планы на будущее

Эта статья продемонстрировала лишь малую часть всех разработок в теории связей, накопленных за несколько лет работы и исследований. В последующих статьях будут

постепенно раскрываться другие проекции теории связей в терминах других теорий, таких как реляционная алгебра, теория графов, а также в терминах теории типов без прямого использования теории множеств, а также анализ отличий от [ассоциативной модели данных Саймона Уильямса \[3\]](#).

Мы также планируем спроектировать теорию связей на саму себя, показав, что ее можно использовать как метатеорию. Это также откроет дверь для проектирования теории множеств и теории типов на теорию связей, то есть мы теперь завершаем цикл определения (теория связей определена в теории множеств, которая сама может быть определена в теории связей). Мы также сможем сравнить теорию множеств, теорию типов, теорию графов, реляционную алгебру и теорию связей, что поможет нам проверить эквивалентность этих теорий или, по крайней мере, получить точную биективную функцию для преобразования между ними.

Также планируется представить ясную и единую терминологию теории связей, её основные постулаты, аспекты и т.д. Текущий прогресс в разработке теории можно отслеживать в [репозитории глубокой теории](#).

Чтобы быть в курсе событий, рекомендуем подписаться на [блог Deep.Foundation](#) здесь, ознакомиться с нашей [работой на GitHub](#) или связаться с нами напрямую в [нашем публичном чате в Telegram](#) (особенно если вы боитесь получить минусы в комментариях).

Мы будем рады любым вашим отзывам, будь то на Хабре, GitHub или Telegram. Вы также можете принять участие в разработке теории или ускорить её развитие, связавшись с нами любым способом.

демонстрация CLI

Теперь вы можете понять, как работает ассоциативная теория, с помощью нашего [демонстрационного инструмента CLI](#), основанного на [нотации связей](#) и хранилище Doublet-ссылок из проекта [LinksPlatform](#). [Нотация связей](#) — это язык для данных, представленных только связями и ссылками. [Дублеты](#) — это движок базы данных, написанный на C# с нуля и поддерживающий только ассоциативное хранение и преобразования.

В этой демонстрации мы используем нотацию связей для создания диалекта, способного описывать одну универсальную операцию — подстановку. Как и в случае с унификацией типов данных, можно объединить создание, чтение, обновление и удаление в одну операцию подстановки. Это похоже на единственную операцию из [алгоритма Маркова](#), который, как доказано, является [полным по Тьюрингу](#).

~ (1.161s)
dotnet tool install --global clink
Tool 'clink' was reinstalled with the stable version (version '2.1.1').

~ (0.289s)
clink '() ((1 1) (2 2))' --changes --after
((1: 1 1)) ((1: 1 1))
((2: 2 2)) ((2: 2 2))
(1: 1 1)
(2: 2 2)

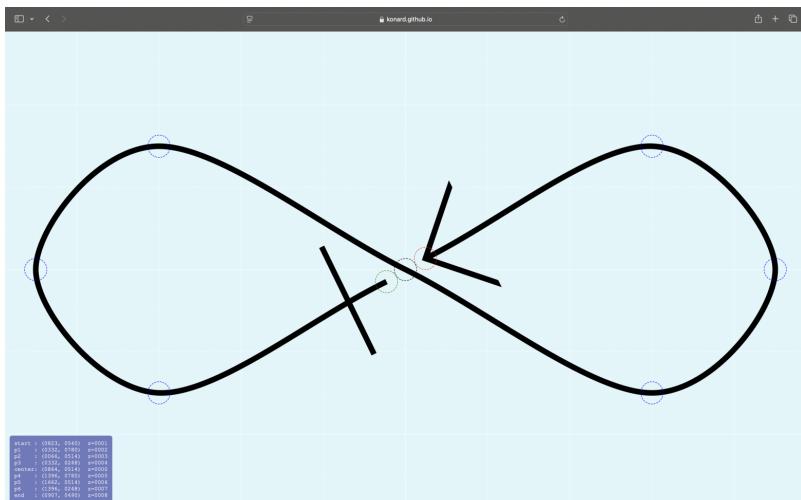
~ (0.137s)
clink '((1: 1 1)) ((1: 1 2))' --changes --after
((1: 1 1)) ((1: 1 2))
(1: 1 2)
(2: 2 2)

~ (0.135s)
clink '(((index: \$source \$target)) (\$index: \$target \$source))' --changes --after
((1: 1 2)) ((1: 2 1))
(1: 2 1)
(2: 2 2)

~ (0.137s)
clink '(* *) ()' --changes --after
((1: 2 1)) ()
((2: 2 2)) ()

Рисунок 11. На этом изображении вы можете видеть создание двух ссылок (1: 1 1) и (2: 2 2); обновление первой ссылки на (1: 1 2); обновление/замену с использованием переменных для обмена источниками и целями каждой ссылки; и удаление всех ссылок с использованием (* *) шаблона.

Визуальные демонстрации



$$L \mapsto L^2$$

Использование LaTeX:

$$L \rightarrow L^2$$

Который отображается как SVG (кликально):

$$L \rightarrow L^2$$

Ссылки

1. Эдгар Ф. Кодд, Исследовательская лаборатория IBM, Сан-Хосе, Калифорния, июнь 1970 г., [«Реляционная модель данных для больших общих банков данных»](#), параграф 1.3, стр. 379
2. Бендер, Эдвард А.; Уильямсон, С. Гилл (2010). «Списки, решения и графы. С введением в теорию вероятностей», раздел 2, определение 6, стр. 161.
3. Саймон Уильямс, Великобритания (1988), [Ассоциативная модель данных](#)
4. Хоман, Дж. В. и Ковач, П. Дж. (2009). [Сравнение модели реляционной базы данных и модели ассоциативной базы данных](#). Вопросы информационных систем, X(1), 208.