logs.md

# 170926/log.md

26.09.2017

# Преобразования в КНФ

Способ Цейтина (с новыми вершинами) против "наивного" метода. Способ Цейтина почти всегда работает лучше, кроме очень маленьких формул. Например, в формуле (a && b) || c лучше раскрыть наивно, а не Цейтином.

# DPLL

DPP плохо годится, чтобы получить выполняющий набор -- он умеет лишь получать логические следствия. Метод резолюций является refutation complete: "если исходная формула противоречива, то противоречие мы точно получим".

DPLL - это алгоритм, который позволяет найти выполняющий набор. Он не на методе резолюций, это просто dfs (backtracking). Сначала заводим формулу в КФН: \Phi = \phi_1 \land \phi_2 \land \dots \phi_n. Далее делаем преобразования:

1. Убрали одинаковые литералы в каждом клозе КНФ (очевидно).
2. Убрали клозы с A \lor \not A (очевидно).
3. Если переменная входит с одним знаком, то можно выкинуть все такие клозы.
4. Unit propagation: клоз из одного литерала {A} --- сразу поняли значение переменной, подставили в остальные клозы (каждый клоз либо уменьшился, либо удалился).
5. Если ничего не помогло, пробуем зафиксировать значение какой-то переменной. По сути просто перебор всех значений с доп.шагами 1-4. Оказывается, что они очень круты и позволяют сильно сократить дерево перебора. Шаги 1-2 можно сделать только в самом начале, поэтому основную работу делают шаги 3-4.

Дополнительные улучшения: а. Выбрать переменную, которая...: входит в максимальное число клозов/входить в самый короткий клоз. б. Какое значение придавать переменной. Привет от Серёжи

# Алгоритм CDCL (conflict-driven clause learning):

Основан на DPLL. DPLL - это хронологический backtracking, он откатывает только значение последней переменной. А CDCL может делать вывод "мы ошиблись со значением переменных несколько выборов назад" и откатиться сразу на несколько шагов назад. А ещё он может запоминать, что какие-то комбинации переменных пробовать не надо.

Идея примерно такая: когда мы получили противоречие (т.е. пустой клоз), то мы смотрим, какие значения переменных привели к обнулению этого клозо. Некоторой структурой данных (implication graph) мы выясняем, из какого условно-минимального набора присваиваний этот клоз обнуляется. Это какая-то конъюнкция условий. Берём отрицание - получаем дизъюнкцию и мы знаем, что она должна быть верна. Просто добавили в исходную формулу. Теперь алгоритм будет очень быстро отсекаться --- это и есть clause learning, переиспользуем в следующих ветках.

Некоторые SAT-солверы делают так: запускают CDCL, обрубают через некоторое время, но выученные клозы запоминают.

# Унификация

Переползаем в область "исчисления первого порядка". По сравнению с пропозициональной логикой у нас появляются: предметные переменные и функциональные значки, предикатные значки, кванторы по предметным переменным. Было в первом семестре.

Лирическое отступление: а логика второго порядка --- это когда мы можем делать кванторы по функциональным значкам.

Чтобы проинтерпретировать формулу, надо выбрать область интерпретации (множество, откуда берутся значения предметных переменных), проинтерпретировать функциональные значки (выбрать всюду определённые функции) и предикатные значки (выбрать всюду определённые предикаты). Обозначается: [[ \forall x . P(x) ]]_I, где I - это интерпретация. I=(D, F, P) --- домен, смысл функциональных, смысл предикатных.

Формулы бывают замкнутые и незамкнутые. Замкнутые всегда либо истины, либо ложны (в зависимости от интерпретации).

Интерпретация замкнутой формулы, при которой она true, называется "моделью".

Выполнимость замкнутой формулы <=> существование модели.

Согласованность интерпретаций: I_1 согласовано с I_2, если есть отображение f: D_1 \to D_2 такое, что значение любого предиката сохраняется при применении f.

Есть term model (Herbrand interpretation): домен "--- это множество термов (константы -- нульарные функции и функциональные значки). Очевидно, что для любой интерпретации I можно подобрать такую интерпретацию значков, что получившаяся интерпретация будет согласована с I.

Обобщение SAT: давайте искать модель для пропозициональной формулы. Из-за Herbrand interpretation нам достаточно лишь подобрать значения предикатов.

<тут-я-уснул-на-моменте-унификации>

Но когда проснулся, ты мы просто взяли алгоритм унификации (как для типов, просто поддерживаем произвольную арность функциональых символов, а не только бинарную стрелочку)

# Обобщаем SAT-солверы на формулы первого порядка

Раньше в методе резолюций мы искали литерал L и \not L и делали резолюцию. А теперь давайте искать одинаковые предикаты (с разными знаками) и унифицировать их. Можно доказать, что от унификации (если она возможна) выполнимость не потеряется. Можем таким образом получить противоречие.

# Хорновские дизъюнкты

Такой клоз - это дизъюнкция литералов, в которые не более одного положительного. Или, если переписать (у которого ровно один положительный): (q_1 \land q_2 \land \dots \land q_k \Rightarrow p).

# Вторая группа пар

Пролог-машина Алгоритм SLD (Selective Linear resolution with Define clause).

## Пример работы

```
add(0, Y, Y).
add(s(X), Y, s(Z)) :- add(X, Y, Z).
 ? add(s(0), s(0), T) # Goal
```

Второе получается хорновским клозом: `add(s(X), Y, s(z)) \lor \lnot(add(X, Y, Z))` .

У нас есть два хорновских клоза, надо вывести goal. Взяли отрицание goal, добавили его как клоз, ищем опровержение методом резолюций. Алгоритм: взяли левый терм из goal, резолюция с кем-нибудь.

## Другой пример

```
add(0, Y, Y).
add(s(X), Y, s(Z)) :- add(X, Y, Z).
? add(s(0), R, s(s(0))
```

Сначала сделали резолюцию со вторым правилом: `[X -> 0, Y -> R, Z -> s(0)]` . Получили `add(0, R, s(0))` . Сматчили, теперь только первое правило (в нём уже новый набор переменных): `[X -> 0, Y -> R, Z -> s(0), Y2 -> R, R -> s(0)]` . Ответ: `R = s(0)` .

### Третий пример

```
add(0, Y, Y).
add(s(X), Y, s(Z)) :- add(X, Y, Z).
? add(s(s(0)), R, s(0)
```

Делаем резолюцию со вторым: `[X -> s(0), Y -> R, Z -> 0]`. Получили `add(s(0), R, 0)`. И вот теперь не смогли сделать никаких резолюций.

### Недетерминированный пример

```
add(s(X), Y, Z) :- add(X, Y, Z).
add(0, Y, Y)
? add(R, s(0), s(s(0))).
```

Сделали резолюцию с первым: `[R -> s(X1), Y1 -> s(0), Z1 -> s(0)]`.

`add(X1, Y1, Z1) ---> add(x1, s(0), s(0))`.

- Сделали с первым (хотя надо бы со вторым): `[X1 -> s(X2), Y2 -> s(0), Z2 -> 0]`. `add(X2, Y2, Z2) ---> add(X2, s(0), 0)`. А теперь не получилось ни с чем сделать. Backtracking.
- Делаем вместо этого со вторым: `[X1 -> 0, Y2 -> s(0)]`. `add(0, Y2, Y2) --> {}` Ура, получилось нужное "ничего". Пишем исходное решение: `R -> s(X1) -> s(0)`.

# 171010a/log.md

# Cut

Cut в прологе --- это восклицательный знак в правой части. Он обрубает перебор в текущей точке. https://stackoverflow.com/a/15065841/767632.

```
max4bad(X, Y, Z) :- le(X, Y), eq(Y, Z).
max4bad(X, _, X).
```

Тут мы попытались написать: если `le(X, Y)`, то выполняется первое, а иначе второе. Но на самом деле второе выполняется всегда. Чтобы добавить это самое "иначе", надо добавить `!`:

```
max4(X, Y, Z) :- le(X, Y), !, eq(Y, Z).
max4(X, _, X).
```

Говорим, что если `le(X, Y)` выполнилось, то в другие ветки ходить никогда не надо. Таким образом, `!` просто обрубает backtracking и ветки перебора между выводом конкретного `max4` и `!`. Обращаем внимание, что он обрубает не вообще весь backtracking, а только некий хвост. Видимо, до последнего решения, которое было принято не при выводе текущего правила (см `02-cut.pl`).

### Семантика negations failure

```
+/(A) :- A,!,fail.
+/(A).
```

Получили этакое отрицание. Если `A` верно, то оно докажется и из-за cut мы сфейлимся. А иначе докажется `+/(A)`, получили "типа отрицание".

Но есть проблема со свободными переменными.

```
p(X) :- eq(X, 123).
```

Если спросить `p(_)`, то будет true, потому что есть переменная: `\exists x: p(x)`. Но если спросить `+/(p(_))`, то будет `false`, несмотря на то, что есть переменная, при которой это верно. Получаем, что на самом деле у нас отрицается вместе с квантором: `\not(\exists x: p(x)) <=> \forall x: \not p(x)`. Итого: нелогичность.

### Predicate completion

Есть интерпретация "predicate completion", в которой отрицание всё-таки разумно:

1. Заменяем все `:-` на `<=>`
2. Если факт не встречается в левых частях, то добавляем его отрицание.

TODO: въехать

# 171010b/log.md

# Гомоморфизмы

Это я пропустил, но тут было что-то с курса по идиомам функционального программирования. Гетерогенные коллекции и иже с ними. https://github.com/dboulytchev/uKanren_transformations/blob/master/src/Term.hs

# miniKanren

Игрушечный язык для реляционного программирования. Язык с неопределённым статусом: Prolog большой, древний, известный, про него все знают и пишут статьи; а вот про miniKanren как-то неоч.

Идея такая: это внешний предметно-ориентированный язык (ориентирован на определённые класс проблем), нахлобучка на другие языки. Самый известный диалект в промышленности: CoreLogic для Closure.

Мотивация для создания miniKanren: вообще говоря, I/O и прочая фигня в логических языках не нужны; на них нужно писать только маленькую логику программы. Поэтому давайте не делать монолитную систему, а сделаем встраиваемый язык.

По сравнению с прологом:

1. Программирование на вашем любимом языке. В рамках этого языка, конечно (нет перегрузки операций/gc --- нам плохо).
2. Язык с полным поиском: найдёт каждое решение, если подождать. До есть не DFS. Но и не BFS --- там хитрее. Interleaving search, кажется, придумал Киселёв(?). Он в начале находит "интересные" термы, а потом "неинтересные".
3. Он гораздо более декларативный. От порядка написания программы зависит меньше.
4. Реляционность --- если есть утверждение вида `P(x, y)`, то можем найти и `x -> y`, и `y -> x` (если есть). В прологе зависит от порядка клозов (но это скорее п.3), можно написать необращаемую формулу (`x = y + 1`, потому что с арифметикой как-то не оч).
5. Не получится оптимизировать программу под конкретную задачу, как на прологе.

### ocanren

Типизированная реализация miniKanren, встроенная в OCaml. Вроде раньше такого никто не делал :) А, скажем, реализация в Java не слишком типизированная (там есть общий тип).

### Основы

Можно делать Goal'ы:

```
t_1 = t_2  // Утверждение о том, что можно унифицировать
(t_1 != t_2)  // Утверждение о том, что нельзя унифицировать
```

Дальше можно делать суперпозиции Goal'ов:

```
G_1 /\ G_2
```

```
G_1 \/ G_2
fresh(x) G  // Введение переменной, квантор существования.
```

## Пример append

Пример на ocanren (пишется почти так):

```
let rec append^o x y xy =
  ((x = []) /\ (y = xy)) \/
    (fresh (h t)
          (x = h :: t) /\
          fresh (ty)
            (xy = h :: ty) /\
                (append^o t y ty)
        )
```

Тип такой: `append : [\alpha] -> [\alpha] -> [\alpha] -> Goal` (каждый список --- считаем, что это терм). Мы написали goal. Дальше можно попробовать вычислять всякую фигню:

```
append [1] q [1; 3]  // Узнаем, что q=[3]
append [1] q [1; r]  // На прологе так нельзя (really?); узнаем, что q=[r]
```

## Пример вывода типом для STLC

STLC (Standard Typed Lambda Calculus) https://github.com/dboulytchev/OCanren/blob/master/regression/test005.ml Вот там записано три правила вывода типов. Можно выводит тип по терму, а можно и наоборот (реляционный-то). Можно вообще дать кусочек терма, типа, и посмотреть, что получится.

## Пример с арифметикой Пеано

Это просто стандартный пример.

## Реляционный интерпретатор

Можно написать реляционный интерпретатор. А дальше он будет по программе и входу генерировать выход. Или по выходу и программе вход. Или по входам и выходам --- программу. Или заполнять кусочки в программе. Или квайн написать, или ещё больший ад. Пример на простом лямбда-исчислении (там, правда, простые ответы): https://github.com/dboulytchev/OCanren/blob/master/regression/test006.ml Охренеть. https://github.com/webyrd/Barliman

Есть мнение, что miniKanren был специально предназначен для написания релияционных интерпретаторов. На прологе успех, конечно, год назад сумели повторить, но им пришлось допиливать сам пролог и его поиск (там же dfs).

## Реляционная сортировка

https://github.com/dboulytchev/OCanren/blob/master/samples/sort.ml Кажется, там написан selection sort.

А потом можно отсортировать список, потом запустить сортировку в обратную сторону и получить все перестановки.

Замечание: miniKanren всё-таки чувствителен к порядку операций. Поэтому при сортировке выгодно написать две версии, которые работают в разные стороны (одна сортирует, другая от-сортирует), но вот одной очень эффективно не получится.

## Про конвертацию программы

Можно конвертировать функциональные программы в реляционные автоматически. Пока что это все пишут руками (тот же интерпретатор), но вроде про это была какая-то работа на конференции (блин, Булычев! А про мой диплом почему так же не получится?). Более того, можно даже с функциями высшего порядка работать.

## Статья

William A. Byrd, ICFP 2017 A unified approach to solve seven programming problems

# 171017/log.md

# Практика

Мы делаем shallow embedding языка Minikanren в Haskell. Без отрицания, правда, оно, типа, сложное. У нас конструкции `===`, `&&&` и `|||` не строят дерево, а потом интерпретируют, а в каком-то смысле сразу вычисляют.

Интересный факт: если написать add вот так:

```
add(x + 1, y, z + 1) :- add(x, y, z).
```

то будет завершаться на x+0=0 после нахождения единственного ответа.

По-хорошему все названия предикатов заканчиваются на 'o' (addo, mulo, appendo).

Кстати, можно попросить сгенерировать список длины три.

# Interleaving search

Сейчас у нас всё написано на мнаде "список", там просто идёт такой DFS. Он может иногда уйти не в ту ветку и не найти решения. Это некруто. В прологе, конечно, тоже DFS, но там у нас есть "cut", который может его контролировать. А в minikanren такого быть не может: он весь такой реляционный.

TODO: Упражение: написать монаду для поиска в ширину. Чем он плох? Тем, что у нас хоть сколько-нибудь длинные ответы оказываются далеко.

Но в Minikanren используется interleaving search. Там, на самом деле, много разных вариантов этого самого search. Самый простой описан в статье про Microkanren (почти не отличается от Minikanren). Это эвристическая штука, которая хорошо себя показывает на интересных примерах (а их всего-то ничего - реляционные интерпретаторы).

Определяем теперь монаду с плюсом `Stream a` (раньше это был список). Ссылка. Три конструктора: пустой список, список с известным первым элементом, невычисленный список.

## Важный момент `mplus`

Что делает `mplus`? Он переставляет по очереди элементы левого и правого аргумента. Таким образом, если у нас один из них был бесконечный, мы всё равно сможем обойти оба. Отсюда прикол: в выражении `mplus a (mplus b c)` у нас на `a` уйдёт половина времени, а на `b` и `c` только четверть.

## Важный момент в `>>=`

Вместо обычного `++` при появлении кучи подцелей мы их не конкатенируем, а переставляем с бывшим хвостом при помощи `mplus`.

## Зачем нужен Mature/Immature?

Чтобы список у нас был реально ленивым. Отличие от Haskell в том, что мы тут можем посмотреть на список и понять, вычислен у него первый элемент или нет, и в зависимости от этого что-то делать. Называется (в контексте minikanren) inverse-eta-delay.

Так можно защищать goal'ы: вместо `g s` можно писать `\s -> Immature (g s)`. Нужна редко. Например, в таком вырожденном примере: `two x = two x ||| (x === s (s o))` Тут interleaving search попробует по очереди смотреть налево и направо. Но чтобы посмотреть налево, надо там найти хотя бы один элемент => бесконечная рекурсия. Поэтому мы должны это дело защитить при помощи `Immature` (встретив его, interleaving search отложит его на следующий раз).

Так как эта штука ничему не мешает, в некоторых реализациях она просто добавляется к каждому Goal.

TODO: упражнение: почему добавление простого `fresh _` не помогает так же, как добавление `Immature`?

# Дальнейшие планы

TODO: упражнение: написать interleaving search.

На следующем занятии: disequality constraint. Говорят, там несложно, но есть техническая работа.

# 171031/log.md

# Disequality constraints

## Лирическое отступление

Важная штука: она есть в MiniKanren, но её нет в Прологе (там только ограниченное). Да и вообще разрешение ограничений (constraints) может быть важнее самого логического программирования. Строго говоря, disequality constraint не увеличивает выразительную сложность языка (как был Тьюринг-полный, так и остался).

## Пример

В MiniKanren disequality constraint (будем называть "дезунификация") вводится на том же уровне, что и унификация, да и выглядит так же. Реализуется несложно, но там есть тонкости. Например, давайте напишем поиск в ассоциативном списке (список пар):

```
let rec lookupo al k v =
    fresh (k' v' t)
            (al === (k' , v') : t) &&&
                ((k' === k &&& v' === v) |||
                 (k' !== k &&& lookupo k t v))
```

Тут нам disequality constraint нужен для того, чтобы выдавать только первый элемент с таким ключом, а не произвольный. Написали на MiniKanren в OCaml код, попросили поискать сколько-нибудь списков, в которых `a` соответствует `b` . Запустили, увидели ответы вида: "список, в котором первые x элементов любые (только ключ не равен `a` ), а элемент x+1 имеет вид `(a, b)` ". На самом деле ответ на задачу, в которой разрешены disequality constraint имеет такой вид: обычный терм, но на некоторые переменные есть дополнительные условия: "она не равна такому-то терму (в котором тоже могут быть подобные ограничения)".

## Алгоритм

Disequality constraint были мутными и непонятными, пока кто-то не догадался, что их можно сделать на основе унификации. Раньше было состояние: (подстановка переменных, номер переменное). Добавляем туда новую информацию: "пул disequality constraint" (это множество подстановок, каждая из которых должна *не выполняться*).

После этого меняем код унификации и добавляем новую операцию "disequality constraint".

1. Меняем унификацию так, чтобы она проверяла, что не нарушились старые disequality constraint. а. Если унифицировать не получилось, то, как и раньше, возвращаем старые ответ. б. Если добавлений к подстановке не произошло, то ничего не поменялось, новой информации нет => ничего не поломалось. в. Если добавилась новая информция phi (это подстановка), то с каждым элементом пула надо сделать: I. Унифицировать его в подстановке phi. Если не получилось - значит, точно уже никогда не получится, всё ок, выкидываем из пула. II. Получилось без дополнительной инфы - упс, уже всегда нарушим, обрубаемся. III. Получилось с дополнительной инфой - заменяем элемент пула на эту дополнительную инфу.
2. Пишем disequality constraint: а. Попытались унифицировать аргументы. Если не получилось, то можно просто ничего не делать. б. Если получилось без новой информации, то он гарантированно нарушен, обрубаемся. в. Если получилось с новой информацией, добавляем её в состояние.

## Сложности с reify

Например, получили ответ: `x` ; в пуле лежит две штуки: `x->y` , `y->x` . И если начнём просто выводить, получим `x[=/=y[=/=x[=/=y[...]]]]` .

### Прочие disequality

В MiniKanren в Racket/Scheme есть ещё три ограничения:

1. numbero - терм является "числом"
2. symbolo - терм является "символом"
3. absento - один терм не встречается в другом

# Встраивание MiniKanren в OCaml

Ocanren - это первое встраивание типизированного minikanren. У нас, например, сейчас написано на Haskell нетипизированно (потому что у всех термов одинаковый тип).

Тут был кошмар какой-то.

# 171107/log.md

# Последнее занятие по MiniKanren

Хочется написать внутри OCanren:

1. Задачу Эйнштейна.
2. Генерилку корректных программ.

Берём посланный код и Makefile с почты, собираем под Linux. Ещё можно посмотреть на сайте OCanren инструкции по сборке его самого, например, там в папке `tests` (или как-то так) есть примерчики, в том числе реляционный интерпретатор.

Если что-то не собирается, можно попробовать выкинуть какую-нибудь зависимость.

# 171114/log.md

# Constraint handling rules (CHR)

Предпоследняя тема нашего курса. Сначала нам даже тему не сказали, но вот задачи, которые надо написать руками:

1. Можем ходить по клетчатой плоскости в четыре стороны (EWNS). Нам дают последовательность действий, мы проходим из исходной точки в конечную. Надо вывести кратчайшую последовательность, которая точно так же доходит из исходной в конечную.
2. Есть монеты в 1, 2, 0.5, 0.2, 0.1 евро. Надо разменять сумму на минимальное количество монет.
3. Есть молекулы водорода ( `H_2` ) и кислорода ( `O_2` ) и правила:

   ```
   1Heat + H_2 + H_2 + O_2 = 2H_2O  // На самом деле взрывается, но пофиг
   1Electricity + 2H_2o = H_2 + H_2 + O_2
   ```

   Нам на вход дают некое количество единиц тепла, электричества и молекул трёх типов, надо просто выполнять реакции, пока можно, в конце вывести, что получилось.

CHR - это логический DSL, который можно реализовать в Java, в Haskell, ещё где-то. Основной реципиент - Prolog. В частности, в SWI-Prolog есть.

В принципе CHR - это частный случай систем переписывания (rewriting systems), которые, увы, нам не рассказали.

### Система переписывания

Пока что говорим про системы переписывания термов. Тогда у нас есть термы (в общем случае - нечто, скажем, граф). Ещё у нас есть сколько-то правил вида `T_1 -> T_2` (в каждом могут быть переменные). В простейшем случае у нас есть один терм (subject), мы перебираем правила, матчим `T_1 с S` , если сматчилось - заменяем на `T_2 с` соответствующими переменными (переменные в `S` , конечно, тоже могут быть, но они там ведут себя как нульарные функции). Точно так же может быть больше одного subject и мы переписываем все вместе независимо.

Пример правила переписывания: `f(X, Y) -> f(Y, X)`.

Это такая система, которая недетерминированно описывает преобразования. Какие потенциально бывают свойства?

1. Терминируемость: не существует бесконечная последовательность переписываний. Обычно её нет без специальных усилий. Но можно доказать не в сильном, а в слабом смысле: множество возможных термов конечно.
2. Конфлюэнтность: если был терм `S`, мы его смогли независимо привести в `A` и в `B`, то обязательно можно найти терм `S'`, в который переводятся `A` и `B`.

Если есть конфлюэнтность и строгая терминируемость, то система переписываний однозначна. Если есть слабая терминируемость и конфлюентность, то система переписываний либо завершится и выдаст единственную нормальную форму, либо будет работать бесконечно.

Одна из первых систем переписывания, по-видимому - лямбда-исчисление.

Есть специальные процедуры, которые умеют проверять, что системы конфлюентны (останавливаются, если да, работают бесконечно, если нет). Можно пробовать добавлять что-нибудь (хз что) в систему, чтобы она стала конфлюентной.

## Основные правила CHR

Для начала бывают constraint таких видов:

1. Интерпретируемые встроенные (builtin) ограничения: например, неравенства целых чисел или что-то такое.
2. Остальные ограничения, задаются семантикой CHR. Но выглядят точно так же, как builtin: просто атомарная формула (функция от чего-то). Всегда есть как минимум два ограничения: true, false.

Ещё можно добавлять свои ограничения двух видов (на самом деле трёх, но пока забьём):

1. Simplification (упрощение:): `[<имя>@] H <=> C | B`
2. Propagation (упрощение:): `[<имя>@] H ==> C | B`

Тут есть:

- `H` - head (голова), мультимножество пользовательских constraint'ов
- `C` - head (голова), мультимножество встроенных constraint'ов
- `B` - head (голова), мультимножество произвольных constraint'ов

Ещё исходно есть мультимножество Goal, в нём лежат constraint'ы. Мы начинаем его переписывать (порядок применения не определён), пока либо не получим неподвижную точку, либо не получим в нём false.

Как переписываются simplification: матчим то, что в `H`, с нашим Goal. Если сматчисло - то проверяем, что `C` выполняются в контексте предположений из Goal. Если выполняются - то выбрасываем из Goal то, что было в `H`, добавляем `B`. В `C`, `B` могут быть связанные с `H` переменные.

Можно ещё добавить правило simpagation, которое выкидывает только часть из `H`:

```
[<имя>@] H \ H1 <=> C | B
```

Но оно заменяется на simplification + propagation, поэтому нас не интересует. В реализациях полезно, не более того.

## Семантика CHR

Пусть у нас был goal `S_1`, он переходит в `S_2`, если:

1. Правило `H <=> C | B`. Переименуем переменные в нём так, чтобы они не пересекались с `S_1`. Тогда если есть такой `H' \subseteq S_1`, то есть подстановка переменных `\sigma`, что `H_{\sigma} = H'`, а `CT |= S_1 => C_{\sigma}` (т.е. в контексте `S_1` по правилам интерпретируемых встроенных constraint'ов верно `C` с подставленными `\sigma`), то `S_2 = S_1 \ H' \cup B_{\sigma}`
2. Аналогично для `H ==> C | B`, только `S_2 = S_1 \cup B_{\sigma}`.

## Логическая семантика CHR

Программа может быть в каком-то смысле (вроде даже не очень творческом, но как хз) представлена как конъюнкция следующих выражений для правил:

1. `H <=> C | B` заменяется на `\forall (C => (H <=> \exists y B(y)))` . Мы говорим, что для всех возможных подстановок переменных в `C` есть равнозначность между `H` с этими переменными и `B` (но в `B` могут возникнуть новые переменные).
2. `H ==> C | B` заменяется на `\forall (C => (H => \exists y B(y)))`

# 171121/log.md

CHR закончили утром (когда меня не было :().

Сейчас идём по книжке "Term Rewriting and all that" Franz Baader, Tobias Nipkow.

Принцип well-founded induction: если у нас есть бинарное отношение `->` на множестве A, то если мы знаем, что для любого x верно следствие "для любого `y` , достижимого из `x` по стрелочкам за конечное число шагов, `P(y)` " => `P(x)` , то `P(x)` верно для произвольного `x` . Сюда зарыты и база, и переход.

Давайте разберёмся, когда этим можно пользоваться. Утверждение: тогда и только тогда, когда `->` терминируемо (т.е. не существует бесконечных цепей). В одну сторону: если терминируемо и посылка верна, но есть `x` , для которого `!P(x)` , то можно применить предположение для него, если не получилось - найти какой-то достижимый из него, снова попробовать, в худшем случае найдём бесконечную цепь. В другую сторону: докажем по индукции " `P(x)` = из `x` нет бесконечных путей".

Теперь занимаемся системами переписывания. Если система нетерминируема, то нам очень-очень плохо и надо стараться сделать её терминируемой.

Повводим каких-нибудь определений конфлюэнтности.

## Локальная конфлюэнтность (local confluence)

Если у нас из `x` есть одна стрелочка в `y_1` и одна в `y_2` , то `y_1 (стрелочка вниз) y_2` , т.е. есть по цепочке из `y_1` и `y_2` в какой-то общий `y'` .

Пример локально конфлюэнтной системы:

```
a <-- 0 <--> 1 --> b
```

Тогда в качестве `x` можем взять либо `0` . У него нет нормальной формы => нет обычной конфлюэнтности. Но есть локальная.

Интуиция: если ориентированный граф в каком-то смысле связен и у него два тупика, то конфлюэнтности наверняка нет. Ещё замечание: эта система нетерминируема.

Теорема: для терминируемых систем локальная конфлюэнтность <=> конфлюэнтность. Доказательство по well-founded induction и локальной конфлюэнтности.

## Строгая конфлюэнтность

Если у нас из `x` есть одна стрелочка в `y_1` и одна в `y_2` , то `y_1 (стрелочка вниз) y_2` , т.е. есть по цепочке из `y_1` и `y_2` в какой-то общий `y'` , причём одна из них длиной не более единицы.

Теорема: докажем, что из строгой конфлюэнтности следует полуконфлюэнтность. Полуконфлюэнтность: если из `x` за одну стрелочку следует `y_1` , а за много - `y_2` , то `y_1<стрелочка вниз>y_2` .

## Diamond property

Если есть одна стрелочка `y_1 <- x -> y_2` , то есть такое `z` , что `y_1 -> z <- y_2` . Из него следует конфлюэнтность. Если транзитивно замкнуть конфлюэнтное отношение, то получим diamond property.

## Какие вообще есть следствия

```
D.P. => C
```

```
LC ===> C (при условии терминируемости)
C ~~~ Church-Rosser
C ~~~ SemiC ~~~ Strong CHR
```

### Замечания

1. Если у нас есть два отношения с одинаковым замыканием, то если в одном есть diamond property, то в другом есть конфлюэнтность.
2. Если есть `->_1 \subseteq ->_2 \subseteq ->_1^*`, то транзитивные замыкания `->_1` и `->_2` равны.

## Системы переписывания

Наблюдение: порядочная система переписывания не вводит свободные переменные при переписывании. То есть в правиле `L -> R` справа нет свободных переменных, которых нет слева.

Дальше мы ввели определение "критической пары". Берём два правила `l_1 -> r_1` и `l_2 -> r_2` (без общих переменных, иначе переименуем). Находим подтерм `l_1`, унифицируем с `l_2`, получили терм `t`. Дальше заменили в `t` `l_1` на `r_1`, а `l_2` на `r_2`, получили два терма (возможно, одинаковые), вот они и образуют критическую пару.

Теорема: система локально конфлюэнтна <=> все критические пары "сводимы". Т.е. их можно, применяя правила, свести к одному.

## 171128/log.md

# Datalog

Это язык, разработанный в 90-х годах. С точки зрения синтаксиса похож на подмножество Prolog. Его можно рассматривать, как Prolog над конечными доменами (домен ~ областями определения переменных).

В чём отличие? В Prolog у нас есть функциональные значки, которые отличаются от предикатов только контекстом (они не могут стоять на самом верхнем уровне - там предикаты). А в Datalog у нас функциональные значки все нульарные, то есть ничего, кроме констант нет => конечные домены.

Предикаты разделяются на два сорта (в теории можно смешивать, но все считают, что они разные):

- Extensional. Это на самом деле предикаты, встречающиеся в командах, которые раньше называлось "фактом". Такая команда, у которой нет правой части (справа от `:-`). В целом в него можно включать переменные, но это необязательно, потому что каждая переменная может принимать лишь конечное число значений.
- Intensional. Это предикаты, которые стоят в левой части от `:-`. Аналогично, в левой части не должно быть свободных переменных.

Алгоритм поиска отличаются от Prolog: мы просто насыщаем (saturate) факты по правилам, пока что-то новое узнаём. Так как все предикаты конечноарные, значений конечно, то это за конечное время выведет всё верное.

И даже можно добавить отрицание. Почти всегда.

## Лирическое отступление: почему отрицание - это плохо

Глобально логическое программирование работает с какими-то потенциально бесконечными/рекурсивными формулами. На них не то чтобы можно "взять и вычислить". Как тогда задавать смысл? Надо делать искать неподвижные точки функций/формулами (привет от Y-комбинатора). Но они не всегда есть.

Есть один прикольный случай, когда всегда есть (теорема Кнастера-Тарского): если у нас есть множество X и частичный порядок-решётка <= на нём, то если какое-нибудь *монотонное* отображение из X на него самого, то у него будет неподвижная точка. А ещё монотонные отображения можно комбинировать, получать монотонное.

На доске была какая-то картинка с X, X^N, X^{X^N} (N-арные функции). Вроде это всё решётки.

Конъюнкция и дизъюнкция монотонны, а вот отрицание не монотонно. Поэтому оно всё ломает.

## Stratified negation

Это мы позволяем в Datalog добавлять отрицание к предикатам, но только иногда. Чтобы не образовались циклы `p :- ... q ...`, `q :- ... !p ...` (возможно, транзитивные). Тогда мы можем взять компонент сильной связности зависимости между предикатами и обработать их в таком порядке. А внутри компоненты предикатов нет. Это чем-то хорошо.

Хорошо тем, что можно просто пополнять набор фактов и не париться про отрицания. А если у нас есть цикл `A -> B -> C`, `!C -> A`, то узнав факт про `A`, мы пополним только `B` и `C`, но вот после этого правило `!C -> A` применять не сможем, ведь мы не знаем, какие из неизвестных фактов C точно неверны, а какие мы просто не вывели.

# Live Variiable Analysis

Пусть есть какой-нибудь простой императивнй язык программирования, в котором есть скалярные переменные (ссылок-указателей нет), можно присваивать в них значения, вычисляемые выражениями, есть `if then goto`, `jmp label`, `stop`. Давайте считать, что у каждой команды есть метка. Ещё считаем, что все переменные изначально проинициализированы.

Можно построить консервативный граф достижимости команд: из какой команды в какую можно перейти (условия `if` не пытаемся проверить, иначе неразрешимая задача). Будем писать, что из команды l достижима `~~>` команда m.

Определим "переменная `x` "жива" в метке `l`": это значит, что есть команда `m`, присваивающая в `x` (`m~~>l`), есть команда `n`, читающая из `x` (`l~~>n`), при этом есть хотя бы один путь `l~~>n`, на котором нет других записей в `x`. Альтернатива: в `x` в метке `l` нельзя ничего записать без изменения семантики программы.

На самом деле условие "была запись раньше" можно выкинуть, потому что переменная исходно инициализирована, иначе будет сложно.

Давайте запишем это на Datalog. Будут предикаты `read(x, l)` (факты), `write(x, l)` (факты), `reach(l, m)` (факты, но транзитивности нет), `live(x, l)` (сейчас запишем).

```
live(x, l) :- read(x, l).
live(x, l) :-
  % тут неявно стоит квантор существования по всем несвязным переменным (m), как обычно.
  live(x, m), reach(m, l), !write(x, l).
  % тут важно, что reach не транзитивен.
```

Возможно, было бы проще записать это дело BFS'ом или чем-то таким. Но зато в виде логического программирования про это дело можно что-то доказывать. Мы же не так просто делаем анализ, а с целью применить оптимизацию.

# BDD (Binary Decision Diagram)

Это такая структура данных для представления логических формул/структур данных. Она весьма компактна и может давать выигрыш по памяти/времени (а может и не давать).

Обычно под BDD подразумевают ROBDD (Reduce Ordered BDD).

Основано на представлении Шеннона: пусть есть пропозициональная формула `F(x_1, ..., x_n)`. Мы можем "разложить" её по переменной `x_i`:

```
F(...) = (x_i && F(x_1, ... 1, ..., x_n)) || (!x_i && F(x_1, ... 0, ..., x_n))
```

Чем оно хорошо: позволяет элиминировать переменную. Например, если `x_i` был в большой-большой конъюнкции/дизъюнкции, то что-то может стереться.

Давайте зафиксируем какой-нибудь порядок элиминации переменных. Элиминировали переменную, получили два поддерева. В каждом тоже элиминировали. Получили такое длинное дерево высоты "размер переменных", на каждом уровне во всех ветках деление по какой-то фиксированной переменной. Это просто BDD, оно ничего не экономит и никому не нужно.

А дальше мы начинаем склеивать уровни снизу вверх: склеили все 0/1 внизу (очевидно, что в листьях уже нет переменных). На следующем уровне склеиваем вершины, у которых совпадают упорядоченная пара детей.

Интересный факт: существуют формулы, для которых можно построить ROBDD быстрее, чем за экспоненту (а ещё это зависит от порядка переменных, который можно подбирать эвристиками). Но есть формулы, для которых всё плохо для любого порядка. Практика показывает, правда, что BDD - это хорошо.

Как это использовать в контексте Datalog: любой элемент конечного домена, можно представлять в виде битового вектора длины n. А предикаты - в виде функций над булевыми векторами, а их - в виде BDD. ВНЕЗАПНО оказывается, что несмотря на то, что константы - это какие-то случайные битовые вектора, то предикаты всё равно в виде BDD хранятся лучше, чем просто как списки множеств.

Ещё факт: на Haskell это либо задолбаться писать, либо будет тормозить. А вот писать на императивном C - самое то: и несложно, и будет выигрыш по времени.

# Ещё про статический анализ кода

Мы уже рассмотрели live variable analysis. Ещё бывают:

- Live Variable Analysis
- Reaching Definitions (RD)
- Available Expressions (AE)
- Upward Exposed Uses (UEU)
- Constant Propagation (CP)

Они все формулируются в терминах read/write в том же микроязыке `C :=`, `goto`, `if then goto`.

Давайте решим первые три на Datalog.

## Reaching Definitions

Назовём "определением `x`" любую операцию, которая присваивает в `x`. Будем обозначать его `x^1` (метка, переменная).

Определим

```
RD(l) = { x^m | x^m ~~> l, при этом на хотя бы одном пути нет записей в `x` }.
```

Это множество позиций, на которых может всё ещё использоваться значение из записи с позиции `l`.

Нахождение такой штуки обычно и подразумевают под "нахождением потока данных". Выглядит просто, но как только у нас появляются указатели, всё становится очень-очень плохо.

Говорят, вмест с AE может использоваться для построения dev-use сеть, потом построить граф совместимости, а потом его покрасить (алгоритм Чайтина) и распределить регистры.

## Available expressions

`sub(E)` - множество всех подвыражений выражения, кроме констант-листьев. Тогда можно говорить, что выражение `E` доступно в точке `l`, если есть такая точка `m~~>l`, что в `m E` было подвыражением, и на любом пути от `m` до `l` нет присваиваний в переменные, которые есть внутри `E`. Вызовов функций у нас, слава богу, нет.

## Upward exposed uses

В каком-то смысле обратная к reaching definitions задача. Ищем, откуда могут быть значения в переменной в данной точке.

## Constant propagation

Если поставили в переменную константу - то её можно попробовать распространить дальше. Для реализации на Datalog не годится, потому что там бесконечная решётка (например, натуральные числа).