1 Описание предметной области

1.1 Логическое и реляционное программирование

Логическое программирование — это вид декларативного программирования, основанный на логике предикатов первого порядка в форме дизъюнктов Хорна (то есть дизъюнктов с только одним положительным литералом), применяющий принципы логического вывода на основе заданных фактов и правил вывода. Программа, написанная на логическом языке — это множество логических формул, выражающих факты и правила, описывающих определённую область проблем.[24]

Существует множество языков логического программирования, таких как Prolog, Curry, Mercurry, однако самые известные — языки семейства Prolog. Prolog применяется для доказательства теорем, проектирования баз знаний, создания экспертных систем и искусственного интеллекта.

Prolog построен на *методе резолюций*, который является обобщением метода “доказательства от противного”, а в частности — на *линейном* методе резолюций (англ. *Linear resolution with Selection function for Definition clauses*). При вычислении программы правило резолюции применяется не к случайных дизъюнктам, а в строго установленном порядке. В случае, когда вычисления дизъюнкта прошло неудачно, происходит *откат* к прошлому состоянию программы, на котором выбирался неудавшийся дизъюнкт[24]. Помимо этого, Prolog вводит разнообразные синтаксические конструкции с побочными *эффектами*, то есть с действиями, приводящими к изменению *окружения* программы, к примеру, оперетор отсечения (англ. *cut*), который влияет на способ вычисления программы.

Описанные выше вещи определяют Prolog, однако из-за них теряется свойство *реляционности*.

Реляционное программирование — это форма чистого логического программирования, в котором программы задаются как набор математических *отношений*.

К примеру, сложение $X + Y = Z$ в терминах реляционного программирования может быть выражено отношением (символ o традиционно использу-

ется для обозначения отношения)

$$\text{add}^o(X, Y, Z),$$

которое в зависимости от того, какие переменные заданы, порождает все возможные значения переменных, при которых отношение выполняется:

- add^o(1, 2, 3) — проверка выполнимости отношения;
- add^o(1, 2, A) — поиск всех таких A, при которых 1 + 2 = A;
- add^o(A, B, 3) — поиск всех таких A и B, при которых A + B = 3;
- add^o(A, B, C) — поиск всех троек A, B и C, при которых A + B = C.

Отношения не предполагают функциональных зависимостей между переменными, поэтому поиск можно проводить в разных “направлениях”, в зависимости от того, какие переменные заданы, как показано выше в примере.

Когда же чистые отношения вырождаются в функциональные и появляется явная зависимость в между переменными, тогда можно говорить про запуск в “прямом” направлении (то есть задаются входные аргументы) и в “обратном” (при задании результата).

К примеру, отношение “меньше” для двух чисел X и Y можно задавать как less^o(X, Y) и получить чистое реляционное отношение, либо как $\text{less}_2{}^o$(X, Y, R), где R сообщает, состоят ли X и Y в отношении, и получить функциональное, и тогда задание X и Y будет прямым направлением, а задание R — обратным.

Реляционная парадигма раздвигает границы логического программирования в своей декларативности и расширяет множество покрываемых им задач.

1.2 miniKanren

miniKanren – семейство встраеваемых предметно-ориентированных языков, специально спроектированное для реляционного программирования[2].

Основная реализация miniKanren написана на языке Scheme[5], однако существует множество встраиваний в ряд других языков, в том числе Clojure, Racket, OCaml, Haskell и другие.

miniKanren предоставляет набор базовых конструкций: унификация ($\equiv$), конъюнкция ($\wedge$), дизъюнкция ($\vee$), введение свежей переменной (fresh), вызов реляционного отношения, — представляющий ядро языка, и разнообразные расширения, к примеру, оператор неэквивалентности (англ. *disequality constraint*) или нечистые операторы, предоставляющие функциональность отсечения из Prolog. Несмотря на то, что в miniKanren введены операторы с эффектами, использование его только с чистыми операторами предоставляет настоящую реляционность.

Классический пример — программа конкатенации двух списков — указан на рисунке 1.

```
1 appendᵒ X Y R =
2   X ≡ [] ∧ Y ≡ R ∨
3   fresh (H X′ R′)
4     (X ≡ H :: X′) ∧
5     (R ≡ H :: R′) ∧
6     appendoᵒ X′ Y R′
```

Рисунок 1 — Пример программы на miniKanren (`::` — конструктор списка)

Пояснение к программе: список R является конкатенацией списков X и Y в случае, когда список X пуст, а Y равен R, либо когда X и R раскладываются на голову и хвост, а их хвосты состоят в отношении конкатенации с Y.

Для выполнения конкатенации над списками необходимо сформировать *запрос* (или *цель*). В запросе в аргументах указываются либо замкнутые термы, либо термы со свободными переменными. Результатом выполнения является список подстановок для свободных переменных, при которых отношение выполняется (когда свободных переменных нет, подстановка, соответственно, пустая).

На рисунке 2 приведёт пример запроса, в котором мы хотим найти возможные значения переменных `Y` и `R`. Потенциально может быть бесконечное число ответов, к примеру, когда все аргументы в запросе — переменные, поэтому в системах miniKanren есть возможность запрашивать несколько первых ответов; в примере, это число 1. Ответы могут содержать в себе как конкретные замкнутые термы (к примеру, числа), так и свободные переменные, которые в примере обозначаются как $\_.n$. В примере одна и также свободная

переменная $\_\cdot_0$ назначена и Y и R. Это означает, что какое бы ни было значение Y, оно всегда будет являться хвостом R.

```
1 > run 1 (Y R) (appendᵒ [1, 2] Y R)
2 Y = _·0
3 R = 1 :: 2 :: _·0
```

Рисунок 2 — Пример запуска отношения конкатенации.

В определение miniKanren входит особый алгоритм поиска ответов — чередующийся поиск (англ. *interleaving search*), основанный на поиске в глубину, который рассматривает всё пространство поиска и гарантирует, что если существует ответ, то алгоритм его предоставит за конечное время[12]. Для сравнения, обычный поиск в глубину, используемый в классическом Prolog при методе резолюций, может зациклиться перед тем, как предоставить все ответы. Это свойство чередующегося поиска определяет, вместе с отсутствием нечистых расширений, реляционность miniKanren.

Хотя miniKanren уже применяется в индустрии для поиска лечения редких генетических заболеваний в точной медицине[18], на данном этапе своего развития используется в основном в исследовательских целях.

Одно из интересных применений miniKanren — *реляционные интерпретаторы*.

Для языка L его интерпретатор – это функция eval_L, которая принимает на вход программу p_L на этом языке, её вход i и возвращает некоторый выход o:

$$\text{eval}_L(p_L, i) \equiv [\![p_L]\!](i) = o,$$

где $[\![\bullet]\!]$ — семантика языка L. Тогда в miniKanren интерпретатор описывается отношением $\text{eval}^o_L(p_L, i, o)$. При запуске такого отношения на разном наборе аргументов можно добиться интересных эффектов:

- по программе p_L и выходу o искать возможные входы i (запуск программы в обратном направлении);
- решать задачи поиска по задаче распознавания [17];
- генерировать программы по заданной спецификации входа i и выхода o (техника программирования по примерам)[3].

TODO: проблемы miniKanren: долгое вычисления сложных алгоритмов, долгие вычисления в обратном направлении.

Одно из возможных решения проблем производительности — специализация.

1.3 Специализация

Специализация программ — это метод автоматической оптимизации программ, при которой из программы удаляются избыточные вычисления, зависимые от частично известного входа. Специализацию программ также называют *частичными* или *смешанными вычислениями*[11].

Специализатор spec_L языка L принимает на вход программу p_L и часть известного входа этой программы i_s (*статических* данных) и генерирует новую программу $\hat{p}_L$, которая ведёт себя на оставшемся входе i_d (*динамических* данных) также, как и оригинальная программа (формула 1).

$$[\![\text{spec}_L(p_L, i_s)]\!](i_d) \equiv \hat{p}_L(i_d) \equiv [\![p_L]\!](i_s, i_d) \tag{1}$$

Специализатор производит все вычисления, зависимые от статических данных, протягивание констант, инлайнгинг и другие.

Одно из интересных теоретических применений специализации — это *проекции Футамуры*[6]. Процесс специализации интерпретатора на программу на языке L $\text{spec}_L(\text{eval}_L, p_L)$ порождает *скомплированную* программу $\hat{p}_L$, а процесс специализации специализатора на интерпретатор языка L $\text{spec}_{L''}(\text{spec}_{L'}, \text{eval}_L)$, в свою очередь, порождает *компилятор*. Это первая и вторая проекции Футамуры соответственно.**TODO: Как-то привязать это**

Специализация разделяется на два больших класса: *online* и *offline* алгоритмы:

- offline-специализаторы — это двухфазовые алгоритмы специализации, в первой фазе которого происходит разметка исхного кода, к примеру, с помощью анализа времени связывания**TODO: cite**, и во второй фазе — непосредственно во время специализации — на основе полученной разметки принимаются решения об оптимизации;
- online-специализаторы, напротив, принимают решения о специализации на лету.

TODO: про то, какие специализаторы когда выгоднее применять (Jones, 147 page) TODO: Примеры

Частичная дедукция — класс методов специализации логический языков[15]. В работе [17] представляется адаптация конъюнктивной частичной дедукции для miniKanren. **TODO: реализация вот это делает хорошо, а вон то — плохо**.

Как будет показано в разделе **??**, конъюнктивная частичная дедукция нестабильно даёт хорошие результаты и может сильно затормозить исполнение программы.

1.4 Суперкомпиляция

Суперкомпиляция — метод анализа и преобразования программ, который отслеживает обобщённую возможную историю вычислений исходной программы и строит на её основе эквивалентную ему программу, структура которой, в некотором смысле, “проще” структуры исходной программы[23].

Это упрощение достигаются путём удаления или преобразования некоторых избыточных действий: удаление лишнего кода, выполнение операций над уже известными данными, избавление от промежуточных структур данных, инлайнинг, превращение многопроходного алгоритма в однопроходный и другие.

Суперкомпиляция включает в себя частичные вычисления, однако не сводится к ним полностью и может привести в глубоким структурным изменениям оригинальной программы.

Суперкомпиляторы, которые используют только “положительную” информацию — то есть информацию о том, что сводобные переменные чему-то равны, — называют позитивными (англ. *positive supercompilation*)[22]. К примеру, при достижении условного выражения **if** x = a **then** t_1 **else** t_2 позитивный суперкомпилятор при вычислении t_1 будет учитывать то, что x = a, однако при вычислении t_2 он не будет знать, что x $\neq$ a. Расширение позитивного компилятора с поддержкой такой “негативной” информации — идеальный суперкомпилятор (англ. *perfect supercompilation*)[20].

Общая схема суперкомпилятора представлена на рисунке 3

История вычислений при суперкомпиляции представляется в виде *графа процессов* — корневого ориентированного графа, в котором каждая ветвь

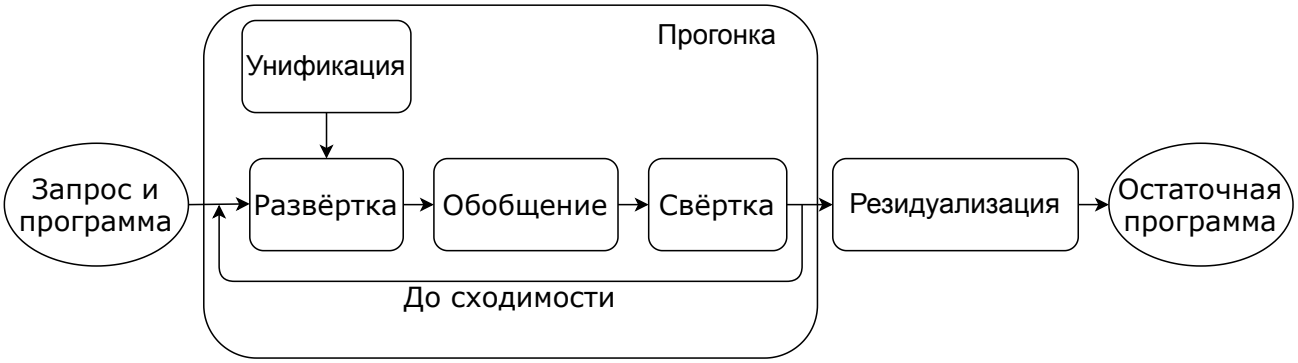

Рисунок 3 — Общая схема суперкомпилятора.

— это отдельный путь вычислений, а каждый узел — состояние системы или *конфигурация*. Конфигурация обобщённо описывает множество состояний вычислительной системы или её подсистемы. К примеру, конфигурацией можно назвать выражение $1 + x$, в котором параметр x пробегает все возможные значения своего домена (допустим, множество натуральных чисел) и задаёт таким образом множество состояний программы.

Процесс построение графа процессов называется *прогонкой* (англ. *driving*). Во время прогонки производится шаг символьных вычислений, после которого в граф процессов добавляются порождённые конфигурации; множество конфигураций появляются тогда, когда ветвления в программе зависят от свободных переменных.

В процессе прогонки в конфигурациях могут появляться новые свободные переменные, которые строятся из исходной конфигурации: если при вычислении выражение его переменная перешла в другую переменную (к примеру, из-за сопоставления с образцом), то в итоговую конфигурацию будет подставлена новая переменная и связь старой и новой сохранится в некоторой *подстановке*. Подстановка — это отображение из множества переменных в множество возможно замкнутых термов. Применение подстановки к выражению заменит все вхождения переменных, принадлежащих её домену, на соответствующие термы.

Пример графа процессов представлен на рисунке **??**. Здесь при исполнении выражение $(a + b) + c$, где a, b, c – натуральные числа, были рассмотрены возможные значения a: это либо оно равно нулю (конструктор Zero), либо это некоторое число a_1, которому прибавили единицу (конструктор Succ). Эти два случая могут задают различные пути исполнения и, соответственно, до-

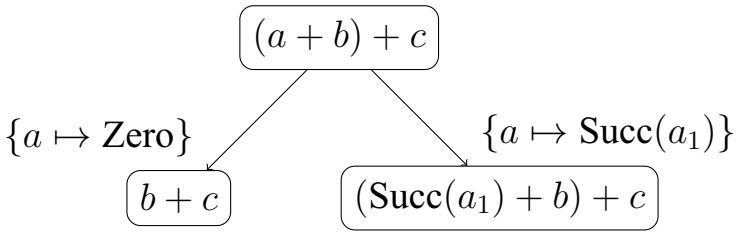

Рисунок 4 — Пример части графа процессов.

бавлены в дерево процессов как два различных состояния, в одно из которых войдёт программа при исполнении.

Потенциально процесс прогонки бесконечный, к примеру, когда происходят рекурсивные вызовы. Для превращения бесконечого дерева вычисления в конечный объект, по которому множно восстановить исходное дерево, используется *свёртка.*

Свёртка (англ. *folding*) — это процесс преобразования дерева процессов в граф, при котором из вершины v_c добавляется ребро в родительскую вершину v_p, если выражение в конфигурации в v_c и в v_p равны с точностью до переименования. Пример ситуации для свёртки изображён на рисунке 6, на котором свёрточное ребро изображено пунтктиром.

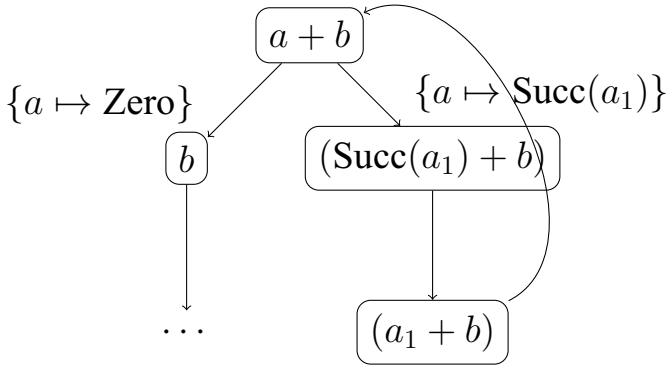

Рисунок 5 — Пример свёртки.

Однако существует ситуации, при котором свёртка не привёт к тому, что граф превратится в конечный объект. Такое может произойти, к примеру, когда два выражения структурно схожи, но не существует переименования, уравнивающих их: $a + b$ и $a + a$.

Для решения этой проблемы используется *обобщение*[21]. Обобщение — это процесс замены одной конфигурации на другую, более абстрактную, описывающую больше состояний программы. Для обнаружения “похожей” конфигурации используется предикат, традиционно называемый *свистком*: сви-

сток пробегает по всем родителям текущей конфигурации и определяет, похожа ли конфигурация на кого-то из них. В случае, когда свисток сигнализирует о найденной схожести, приеняется обобщение. Сам шаг обобщения может произвести дейтсвия трёх видов:

– *обобщение вниз* приводит к тому, что новая конфигурация заменяет текущую в графе процессов;

– *обобщение вверх* приводит к замене родительской конфигурации на обобщённую;

– *разделение* (англ. *split*) используется для декомпозиции выражений, которые затем будут обработаны отдельно.

Пример обобщения представлен на рисунке **??**

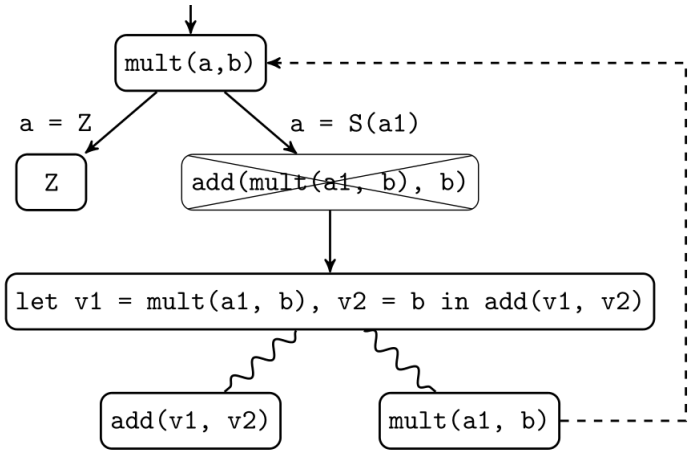

Рисунок 6 — Пример обобщения.

Построение программы по графу конфигураций называется *резидуализацией*, а построенная программа — *остаточной* (англ. *residual*). Алгоритм выявления остаточной программы основан на обходе дерева, но в остальном полностью зависит от языка.

TODO: Написать ещё про СК; продемонстрировать результаты СК; выводы

2 Специализация miniKanren

В данной работе для специализации был выбран μKanren — минималистичный диалект языка miniKanren[8]. **TODO: todo**

Абстрактный синтаксис языка представлен на Рисунке 7.

$\mathcal{C}$	$=$	$\{C_i\}$	конструктор с арностью i
$\mathcal{X}$	$=$	$\{x, y, z, \dots\}$	переменные
$\mathcal{T}_X$	$=$	$X \cup \{C_i(t_1, \dots, t_i) \mid t_j \in \mathcal{T}_X\}$	термы над множеством переменных X
$\mathcal{D}$	$=$	$\mathcal{T}_\emptyset$	замкнутое выражение
$\mathcal{R}$	$=$	$\{R_i\}$	реляционный символ с арностью i
$\mathcal{G}$	$=$	$\mathcal{T}_\mathcal{X} \equiv \mathcal{T}_\mathcal{X}$	унификация
		$\mathcal{G} \wedge \mathcal{G}$	конъюнкция
		$\mathcal{G} \vee \mathcal{G}$	дизъюнкция
		$\texttt{fresh}\ \mathcal{X}\ .\ \mathcal{G}$	введение свежей переменной
		$R_i(t_1, \dots, t_i),\ t_j \in \mathcal{T}_\mathcal{X}$	вызов реляционного отношения
$\mathcal{S}$	$=$	$\{R_i^j = \lambda\ x_1 \dots x_i\ .\ g_j;\}\ g$	спецификация программы

Рисунок 7 — Синтаксис языка μKanren [19].

– Унификация двух термов $t_1 \equiv t_2$ порождает подстановку θ, называемую *унификатором*, такую что её применение к термам уравнивает их: $t_1\theta = t_2\theta$.

  Алгоритм унификации языков семейства miniKanren использует проверку вхождения (англ. *occurs check*), что гарантирует корректность получаемых унификаторов, однако довольно сильно замедляет выполнение программ.

– Конъюнкция двух целей $g_1 \wedge g_2$ подразумевает одновременное успешное выполнение выражений g_1 и g_2.

– Дизъюнкция двух целей $g_1 \vee g_2$ подразумевает, что достаточно, чтобы хотя бы одно из выражений g_1 или g_2 выполнялось успешно. Следует отметить, что при выполнении g_1 выражение g_2 также будет вычисляться.

– Введение свежей переменной fresh $x\ .\ g$ в языках miniKanren нужно указывать явно, в отличие, к примеру, от Prolog, где это происходит неявно.

– Вызов реляционного отношения приводит к тому, что переданные в отношение термы унифицируются со аргументами отношения и подставляются в тело отношения.

В контексте вычислений важно различие между *синтаксическими* переменными, которые определяются в тексте программы и обычно представляют-

ся строковыми литералами, и *семантическими* переменными, которые непосредственно используются в процессе вычислений и представляются целыми числами, с которыми легче работать и генерировать свежие.

TODO: Нужно что-то ещё написать

2.1 Библиотека специализации

Реализация суперкомпилятора для μKanren строилась на основе проекта по специализации μKanren с помощью конъюнктивной частичной дедукции[1] на функциональном языке программирования Haskell. Результаты специализации μKanren представлены в работе [17].

Библиотека вводит ряд структур данных и алгоритмов для реализации конъюнктивной частичной дедукции, однако существует возможность её переиспользования для суперкомпиляции в силу нескольких доводов:

- схожесть методов частичной дедукции и суперкомпиляции[7], из чего следует, что ряд вспомогательных функций и определений и для частичной дедукции, и для суперкомпиляции будут совпадать;
- алгоритм обобщения конъюнктивной частичной дедукции[4], о котором подробнее будет рассказано позже, имеет ряд общих черт с шагами обобщения в суперкомпиляции;
- библиотека предоставляет возможность преобразования сгенерированной программы на miniKanren, при котором удаляются излишние унификации и происходит удаление излишних аргументов (англ. *redundant argument filtering*), что приводит в увеличению производительности программы, поскольку унификация операция дорогая.

В библиотеке введены структуры данных, описывающие термы и выражения языка в соответствии с синтаксисом μKanren на рисунке 7. Над ними введён ряд важных структур и алгоритмов, речь о которых пойдёт ниже. Основные операции над выражениями в конъюнктивной частичной дедукции производятся над конъюнкциями *атомов*, — то есть неделимыми элементами, которыми в miniKanren являются вызовы реляционных отношений.

[1] `https://github.com/kajigor/uKanren_transformations/`

Во-первых, реализован алгоритм унификации двух термов. Операция унификации находит наиболее общий унификатор (англ. *most general unifier*), причём единственный[14], то есть такой унификатор θ, что для любого другого унификатора θ' существует подстановка σ, с которой композиция наиболее общего унификатора даёт θ': $\theta' = \sigma \circ \theta$[14]. К примеру, для двух термов $f(X, 2)$ и $f(1, Y)$ наиболее общим унификатором является подстановка $\{X \mapsto 1, Y \mapsto 2\}$, когда подстановки вроде $\{X \mapsto 1, Y \mapsto Z, Z \mapsto 2\}$ также унифицирует термы, однако содержат в себе лишние элементы. Поиск наиболее общего унификатора уменьшает размер итоговой подстановки и является предпочтительным.

Во-вторых, реализованы предикаты над термами, которые проясняют описанные ниже возможные связи термов.

- Выражение e_2 является *экземпляром* выражения e_1 ($e_1 \preccurlyeq e_2$) если существует такая подстановка θ, применение которой приравнивает два выражения $e_1\theta = e_2$; также говорят, что e_1 более общий, чем e_2. К примеру, $f(X, Y) \preccurlyeq f(Y, X)$ и $f(X, Y) \preccurlyeq f(Y, X)$.
- Выражение e_2 является *строгим* экземпляром выражения e_1 ($e_1 \prec e_2$), если $e_1 \preccurlyeq e_2$ и $e_2 \not\preccurlyeq e_1$. К примеру, $f(X, X) \prec f(X, Y)$, но не наоборот.
- Выражения e_1 и e_2 *варианты* друг друга $e_1 \approx e_2$, если они являются экземплярами друг друга.

Предикат над вариантами по своей сути определяет, являются ли два терма переименованием друг друга, и поэтому может быть использован для свёртки графа процессов. Предикаты над экземплярами определяют схожесть термов и используется в обобщении и в алгоритмах суперкомпиляции [22].

В-третьих, в качестве свистка используется отношение *гомеоморфного вложения*[21]. Отношение гомеоморфного вложения $\trianglelefteq$ определено индуктивно:

- переменные вложены в переменные: $x \trianglelefteq y$;
- терм X вложен в конструктор с именем C, если он вложен в один из аргументов конструктора:

$$X \trianglelefteq C_n(Y_1, \ldots, Y_n) : \exists i, X \trianglelefteq Y_i;$$

– конструкторы с одинаковыми именами состоят в отношении вложения, если в этом отношении состоят их аргументы:

$$C_n(X_1, \ldots, X_n) \trianglelefteq C_n(Y_1, \ldots, Y_n) : \forall i, X_i \trianglelefteq Y_i.$$

К примеру, выражение $c(b) \trianglelefteq c(f(b))$, но $f(c(b)) \not\trianglelefteq c(f(b))$. Преимущество использования гомеоморфного вложения, в первую очередь, состоит в том, что для этого отношения доказано, что на бесконечной последовательности выражений $e_0, e_1, \ldots, e_n$ обязательно найдутся такие два индекса $i < j$, что $e_i \trianglelefteq e_j$, вне зависимости от того, каким образом последовательность выражений была получена [22]. Это свойство позволяет доказать завершаемость алгоритм суперкомпиляции.

Однако отношение гомеоморфного вложения допускает, чтобы термы $f(X, X)$ и $f(X, Y)$ находились в отношении гомеоморфного вложение $f(X, X) \trianglelefteq f(X, Y)$ в силу того, что все переменные вкладываются друг в друга. Однако в приведённом примере обобщение $f(X, X)$ и $f(X, Y)$ не привело бы к более общей конфигурации.

Отношение *строгого* гомеоморфного вложения $\trianglelefteq^+$ вводит дополнительное требование, чтобы терм X, состоящий в отношении с Y, не был *строгим экземпляром* Y[16]. В таком случае отношение $f(X, X) \not\trianglelefteq^+ f(X, Y)$, поскольку $f(X, Y)$ является строгим экземпляром $f(X, X)$ из-за того, что существует подстановка $\{X = X, Y = X\}$.

В рамках конъюнктивной частичной дедукции понятие гомеоморфного вложение было расширено на конъюнкции выражений. Пусть $Q = A_1 \wedge \cdots \wedge A_n$ и Q' — конъюнкции термов, тогда $Q \trianglelefteq Q'$, тогда и только тогда, когда $Q' \neq Q$ и существует упорядоченные подконъюнкции $A'_1 \wedge \cdots \wedge A'_n$ конъюнкции Q' (то есть Q' может содержать больше выражений, чем Q), такие что $A_i \trianglelefteq A'_i$. Конъюнкция Q' может содержать в себе больше выражений за счёт того, что в этом случае при обобщении производёт шаг разделения. Это расширение было реализовано в рассматриваемой библиотеке.

В-четвёртых, реализован алгоритм обобщения для конъюнктивной частичной дедукции. В общем, алгоритмы обобщения основаны на понятии *наиболее тесного обобщения*.

– *Обобщение* выражения e_1 и e_2 — это выражение e_g, такое что $e_g \preccurlyeq e_1$ и

$e_g \preccurlyeq e_2$. На пример, обобщением выражения $f(1, Y)$ и $f(X, 2)$ является $f(X, Y)$.

– Наиболее тесное обобщение (англ. *most specific generalization*) выражений e_1 и e_2 — это обобщение e_g, такое что для каждого обобщения $e'_g \preccurlyeq e_1$ и $e'_g \preccurlyeq e_2$ выполянется $e'_g \preccurlyeq e_g$[22]. Функция обобщения принимает на себя два терма t_1 и t_2 и возвращает тройку $(t_g, \theta_1, \theta_2)$, такую что $t_1\theta_1 = t_g$ и $t_2\theta_2 = t_g$, при этом θ_1 и θ_2 назовём *обобщающими унификаторами* (англ. *generalizers*).

Алгоритм для конъюнктивной частичной дедукции , в соответствии со своим определением гомеоморфного вложения, выберет предка Q^p для конъюнкции Q, такого что $Q^p \trianglelefteq^+ Q$, и **TODO: вообще не понятно что**

Этот алгоритм используется в рамках конъюнктивной частичной дедукции для построения более сложных алгоритмов, свойственных методам частичной дедукции. Однако его можно использовать как самостоятельный алгоритм для суперкомпиляции, который соединяет в себе возможность произвести шаги обобщения и разделения вместе, не выделяя отдельные шаги для это в процессе прогонки.

2.2 Реализация суперкомпиляции

Представление графа процессов в Haskell затруднено тем, что графовые структуры данных обычно требуют ссылок на произвольные узлы, что приводит к появлению перекрёсных ссылок. Прямая реализация этой идеи сложна в разработке и поддержке и не является идиоматичной. Использование *IORef*[2], хотя и предоставляет мутабельность, приводит к неоправданному усложнению кода всего проекта, избавляя код от функциональной чистоты.

Заметим, что графовость этой структуре данных придают *обратные рёбра* (то есть рёбра от детей к родителям), которые появляются при свёртке, когда ребёнок является переименованием родителя. Тогда, если уметь сохранять или восстанавливать информацию об этой связи, то достаточным будет представить граф в качестве *дерева* процессов. Древовидные структура однозначно отображается на процесс символьных вычислений, а также они легко и идиоматично в Haskell.

[2]https://hackage.haskell.org/package/base-4.11.1.0/docs/Data-IORef.html

Структура дерева процессов представлена в виде дерева на рисунке 8.

```
 1
 2 type Conf = Conjunction (RelationCall FreeVar)
 3
 4 type Subs = Variable ↦ Term
 5
 6 data Tree where
 7   Failure     :: Tree
 8   Success     :: Subst → Tree
 9   Renaming    :: Conf → Subst → Tree
10   Abstraction :: Conf → Subst → List Tree → Tree
11   Generalizer :: Subst → Tree → Tree
12   Unfolding   :: Conf → Subst → List Tree → Tree
```

Рисунок 8 — Описание дерева процессов.

Конфигурация `Conf` определена как выражение со свободными переменными. В узле дерева процессов хранится конфигурация, приведённая к форме, содержащей только конъюнкцию вызовов реляционного отношения. Это сделано из тех соображений, что, во-первых, дизъюнкция представляет собой ветвление вычислений, посему, соответственно, представляется как ветвление в дереве процессов, во-вторых, унификации производятся во время символьных вычислений и добавляются в подстановку, в-третьих, так как введение свежей переменной оказывает влияние лишь на состояние, в котором производятся вычисления, неосмысленно сохранять его в конфигурации.

Подстановка `Subst` соответствует своему математическому определению как отображению из переменных в термы. Узлы дерева процессов представляют шаги суперкомпиляции и исходы вычисления выражений:

- `Failure` обозначает неудавшееся вычисления. Такой исход случается при появлении противоречивых подстановок;
- `Success`, напротив, обозначает удавшееся вычисление, которое свелось к подстановке `Subst`;
- `Renaming` обозначает узел, конфигурация которой является переименованием какого-то родительского узла.
- `Abstraction` обознает узел, который может быть обобщён на одного из родителей. После обобщения может появится несколько конфигура-

ций, которые являются результатом применения разделения. Эти конфигурации добавляются в качестве списка дочерних поддеревьев в текущий узел.

– `Generalizer` хранит себе унификатор, который порождается во время обобщения двух термов, и поддерево с обобщённой конфигурацией.

– `Unfolding` обознает шаг символьного вычисления, на котором произошёл шаг вычислений и по рассматриваемой на этом шаге конфигурации породились новые конфигурации.

Окружение для суперкомпиляции должно сохранять следующие объекты:

– подстановку, в которой содержатся все накопленные непротиворечивые унификации, необходимую в процессе прогонки для проверки новых унификаций;

– первую свободную семантическую переменную, необходимую для генерации свежих переменных, к примеру, при абстракции;

– определение программы, необходимое для замены вызова на своё тело.

TODO: Что-то ещё об этом нужно написать!

Шаг символьного вычисления по данной конфигурации C порождает множество конфигурации $\{C_1, \ldots, C_n\}$, описывающих состояния в которое может перейти процесс реального исполнения программы. Классически, шаг символьного вычисления соответствует семантике языка, который суперкомпилируется, и для μKanren существует сертифицированная семантика[19], однако описание шага символьного вычисления μKanren для суперкомпиляции усложнено тем, что реляционные языки не исполняются привычным образом, как, к примеру, функциональные программы, и *поиск*, вшитый в семантику, не ложится на суперкомпиляцию.

Тогда порождённую конфигурацию можно рассматривать не как непосредственный шаг вычисления, но как возможное состояние, в которое может перейти программа. Такое состояние появляется путём раскрытия тела одного или нескольких конъюнктов конфигурации.

К примеру, рассмотрим часть программы на μKanren на рисунке 9, в котором определены какие-то отношения `f` и `g`.

```
1 f(a) = f'(a)∨f''(a)
2 g(a, b) = g'(a)∧g''(b)
```

Рисунок 9 — Пример отношений для демонстрации шага символьных вычислений

Допустим, на шаге суперкомпиляции алгоритм обрабатывает конфигурацию $f(v_1) \wedge g(v_1, v_2)$ хотим сделать шаг символьного вычисления. Рассмотрим несколько способов породить новые конфигурации.

– Если раскроется определение f, то будут получены новые конфигурации $f'(v_1) \wedge g(v_1, v_2)$ и $f''(v_1) \wedge g(v_1, v_2)$.
– Если раскроется определение g, то будет получена новая конфигурация $f(v_1) \wedge g'(v_1) \wedge g''(v_2)$.
– Если раскроются оба определения f и g, то будут получены новые конфигурации $f'(v_1) \wedge g(v_1) \wedge g''(v_2)$ и $f''(v_1) \wedge g(v_1) \wedge g''(v_2)$.

Последний набор конфигураций — это полный набор состояний, в которые процесс вычислений может прийти. В первых двух наборах, можно отметить, порождённые конфигурации не исключают возможные состояния процессов, отображённые в последнем наборе, они могут появится на последующих шагах вычисления, если перед этим ветвь исполнения не будет остановлена из-за противоречивой подстановки.

Таким образом, какой-бы способ развёртывания определений не был бы выбран, он не будет исключать состояния, в которые процесс вычисления теоретически может прийти, но выбор разных стратегий развёртывания может систематически приводить к разным деревьям процессов, а следовательно, приводить к различным эффектам специализации.

Базовой стратегией порождения новых конфигураций выбрана *полная стратегия развёртывания*, пример которой был показан выше, при которой мы заменяем определния всех реляционных вызовов конфигурации.

В суперкомпиляции, в отличие от методов частичной дедукции, в обобщение включён шаг обобщения вверх, при котором происходит не подвешивание обобщённой конфигурации в качестве потомка конфигурации, которая обобщалась, но замена самого родителя на новую конфигурацию, поддерево же родителя уничтожается. Для определения необходимости обобщать вверх

введём предикат $e_1 \lessdot e_2$, который определяет, что $e_1 \prec e_2$ и $e_2 \not\prec e_1$. Такое ограничение необходимо из-за того, что суперкомпилятор оперирует конъюнкциями выражений и делает операции разделения и обобщения вниз за один шаг с конъюнкциями возможно разной длины, однако для обобщения вверх необходимо удостовериться, что одни **TODO: todo**

2.2.1 Обобщённый алгоритм суперкомпиляции

На основе введёных выше терминов и операторов можно составить обобщённый алгоритм суперкомпиляции, который не затрагивает особенности и трудности реализации на Haskell. Обобщённый алгоритм суперкомпиляции на псевдокоде представлен на рисунке 10.

```
 1 drive(env, tree, configuration):
 2   if configuration is empty
 3   then add(env, tree, success node)
 4   else if ∃ parent: configuration ≈ parent
 5   then add(env, tree, renaming node)
 6   else if ∃ parent: parent ⋖ configuration
 7   then
 8      node ← generalize(configuration, parent)
 9      addUp(env, tree, parent, node)
10   else if ∃ parent: parent ⊴⁺ configuration
11   then
12     add(env, tree, abstraction, node)
13     children ← generalize(configuration, parent)
14     ∀child ∈ children :
15       drive(env, tree, child)
16   else
17     add(env, tree, unfolding node)
18     children ← unfold(env, configuration)
19     ∀child ∈ children :
20       drive(env, tree, child)
```

Рисунок 10 — Обобщённый алгоритм суперкомпиляции.

2.2.2 Конкретный алгоритм суперкомпиляции

Наличие операции обобщения вверх предполагает, что необходимо умение передвигаться по дереву вверх и изменять его. Реализация в Haskell этой

идеи — задача крайне нетривиальная. Возможно представлять деревья в мутабельных массивах, однако при обобщении необходимо удалять целые поддеревья, что при таком подходе сложная операция.

Классическим способом решения этой проблемы являются *зипперы*[10]. Эта идиома предлагает рассматривать структуру данных как пару из элемента, на котором установлен фокус, и контекста, который представляется как структура данных с “дыркой”, в котором сфокусированный элемент должен находиться.

К примеру, зиппер для списка `[1, 2, 3, 4]` при фокусе на 3 представляется таким образом: `(3, ([2, 1, 0], [4, 5, 6]))`. Тогда перефокусировка вправо или влево на один элемент происходит за константу, как и замена элемента, для которой достаточно заменить первую компоненту пары. В то время как, в силу того, что операция взятия элемента в связном списке по индексу происходит за линейное время от длины списка, взятие элемента слева от 3 также будет происходить за линейное время, как и, соответственно, модификация списка.

Для деревьев с произвольным количеством детей зиппер может выглядеть как пара из текущего узла и списка родителей, отсортированного в порядке близости к узлу (рисунок 11).

```
1 data Parent = Parent { children :: ListZipper Node }
2 type TreeZipper = (Node, List Parent)
```

Рисунок 11 — Пример структуры зиппера для деревьев

Родительский (структура `Parent`) список детей представлен в виде зиппера (поле `children`) для списка, в котором происходит фокус: у непосредственного родителя — на элемент в фокусе, а у остальных родителей — на предыдущего в порядке сортировки.**TODO: more?**

При представлении дерева процессов в идиоме зипперов основа алгоритма суперкомпиляции принимает форму описания действий при смене состояния зиппера. **TODO: TODO**

2.2.3 Модификации базового алгоритма суперкомпиляции

Поиск узлов для переименования среди всех вычисленных поддеревьев

В базовом алгоритме суперкомпиляции поиск узлов на которые происходят переименования происходит среди родителей. Это напрямую соотносится с понятием символьных вычислений: по достижении узла, которое является переименованием уже встреченного, вычисление переходит на родительский узел. Однако довольно части встречается, что в разных поддеревьях дерева процессов стречаются одинаковые конфигурации, поддеревья которых оказываются полностью идентичными. В таком случае, кажется очевидной оптмизация, при которой мы запоминаем вычисленные поддеревья и в случае, когда мы встречаем схожу конфигурацию, не вычисляем поддерево заново, добавляя ссылку на него. **TODO: Показать, почему это ничего не сломает**

Стратегии развёртывания реляционных вызовов

Как уже говорилось, разные стратегии развёртывания реляционных вызовов могут привести к разным эффектам специализации. К примеру, полная стратегия развёртывания, которая была принята за базовую, приводит к *манлингу* (англ. *tupling*)[9] — оптмизации, при которой множество проходов по одной структуре данных заменяется на один проход.

Основной недостаток базового подхода в том, что он для получения всех возможных состояний производит декартово производение тел вызовов в конъюнкциях, что приводит к сильному разрастанию дерева процессов и, как следствия, сильно требователен к вычислительным ресурсам. Вследствие чего реализация новых стратегий развёртывания производится не только в исследовательских, но и прикладных целях.

Для лёгкой подмены стратегий суперкомпиляции был разработан специальный интерфейс `UnfoldableGoal` (рисунок 12).

```
1 class Unfoldable a where
2   initialize :: Conf → a
3   get        :: a → Conf
4   unfoldStep :: a → Env → List (Env, a)
```

Рисунок 12 — Интерфейс для различных стратегий развёртывания.

Предоставляемые интерфейсом функции используются в алгоритме суперкомпиляции следующим образом:

– `initialize` оборачивает конфигурацию в структуру, в которой может содержаться вспомогательная информация для процесса развёртывания;

– `get` позволяет получить конфигурацию для применения её к операциям, не зависящим от стратегий;
– `unfoldStep` непосредственно проводит шаг вычисления на основе текущей конфигурации и её окружения, порождая новые конфигурации с соответствующими им состояниями.

В работе рассмотрен и реализован ряд стратегий, описанные ниже.

– **Последовательная стратегия развёртывания**, при которой отслеживается, какой вызов был раскрыт на предыдущем шаге, чтобы на текущем раскрыть следующий за ним.**TODO: todo**
– **Нерекурсивная стратегия развёртывания**, при которой в первую очередь раскрывается нерекурсивный вызов в конфигурации. Нерекурсивность определяется лишь тем, содержит ли определение реляционный вызов самого себя. Более сложный анализ структуры функций не мог бы быть использован в силу того, что тогда было бы необходимо реализовать класс алгоритмов анализа, что совершенно отдельная задача.

  Ожидается, что при нерекурсивной стратегии развёртывания из конфигураций будут как можно быстрее появляться выражения, которые могут быть сокращены или вовсе удалены из-за унификации (к примеру, отношения, кодирующие таблицы истинности, такие как and^o) или привести к скорой свёрте. **TODO: todo**.
– **Стратегия развёртывания вызовов с минимальным количеством ветвлений**, при которой на каждом шаге вычисления будет появляться минимально возможное количество конфигураций, что приведёт к минимальной ветвистости дерева.

Частичный или полный отказ от обобщения вверх

Обобщение вверх приводит к тому, что происходит замена целого поддерева процессов предка, на которого обобщается конфигурация. Иногда это может приводить к тому, что теряются аргументы частично известного входа. К примеру, на рисунке 13 представлено дерево процессов, при котором происходит обобщение вверх. **TODO: пояснения к примеру**

В случае же, когда обощение происходит на сам корень дерева, теряется эффект протягивания констант.

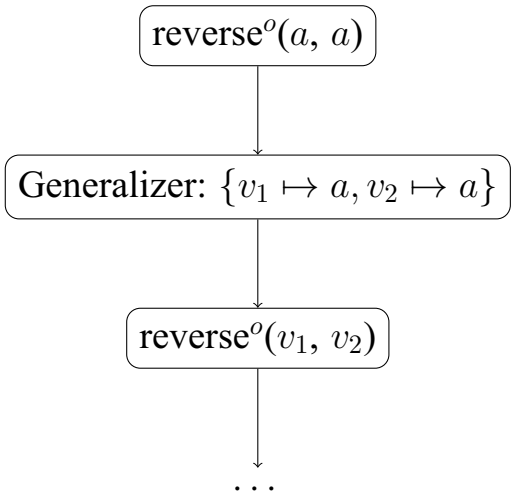

Рисунок 13 — Демонстрация потери информации при обобщении вверх.

Обобщение на все вычисленные узлы, не только на родительские
TODO: Понять, почему это не противоречит методам суперкомпиляции

Расширение языка μKanren с помощью операции неэквивалентности

Множество операции в оргинальном μKanren покрывает все нужды реляционного программирования, однако на ряде программа оно вычислительно допускает пути исполнения, которые не приводят к успеху, однако сообщить об этом не представляется возможным.

К примеру, на рисунке 14 изображена операция поиска значения по ключу в списке пар ключ-значения lookup^o.

```
1 lookupᵒ K L R =
2    (K', V) :: L' ≡ L ∧
3    (K' ≡ K ∧ V ≡ R ∨ lookupᵒ K L' R)
```

Рисунок 14 — Отношения поиска значения по ключу.

В соответствии с программой список `L` должен иметь в голове пару из ключа и значения `(K', V)` и либо этот ключ `K'` унифицируется с искомым ключом `K` и значение `V` — с результатом `R`, либо поиск происходит в хвосте списка `L'`. Проблема этой программы в том, что если унификация `(K',V)::L' ≡ L` прошла успешно и был найден результат, то поиск всё равно продёт во вторую ветку с рекурсивным вызовом и будет искать значение дальше, хотя по семантике поиска ключа в списке должен вернуться

лишь одно значение. Более того, суперкомпилятору тоже придётся учитывать и, возможно, проводить вычисления, которые не принесут никакой пользы.

В miniKanren существует операция неэквивалентности $t_1 \not\equiv t_1$, вводящее ограничение неэквивалентности (англ. *disequality contraints*)[1]. Операция неэквивалентности определяет, что два терма t_1 и t_2 никогда не должны быть равны, накладывая ограничения на возможные значения свободных переменных терма.

Расширение синтаксиса μKanren представлено на рисунке 15.

$$\begin{array}{lll} \mathcal{G} = & \ldots & \\ & \mathcal{T}_{\mathcal{X}} \not\equiv \mathcal{T}_{\mathcal{X}} & \text{дезунификация} \end{array}$$

Рисунок 15 — Расширение синтаксиса μKanren относительно указанного на рисунке 7.

Исправленная версия отношения lookup^o представлена на рисунке 16.

```
1 lookup^o K L R =
2    (K', V) :: L' ≡ L ∧
3    (K' ≡ K ∧ V ≡ R ∨
4     K' ≢ K ∧ lookup^o K L' R)
```

Рисунок 16 — Исправленное отношение поиска значения по ключу.

В такой реализации две по сути исключающие друг друга ветви исполнения будут исключать друг друга и при вычислении запросов, и при суперкомпиляции.

Для реализации ограничения неэквивалентности вводится новая сущность под названием “хранилище ограничений” Ω (англ. *constraints store*), которое используется для проверки нарушений неэквивалентности. Окружение расширяется хранилищем ограничений, которое затем используется при унификации и при добавлении новых ограничений.

Тогда нужно ввести следующие модификации в алгоритм унификации конфигурации, который собирает все операции унификации в конъюнкции перед тем, как добавить её в множество допустимых конфигураций.

– При встрече операции дезунификации $t_1 \not\equiv t_2$ необходимо произвести следующие действия. Применить накопленную подстановку к термам

$t_1\theta = t'_1$ и $t_2\theta = t'_2$ и унифицировать термы t'_1 и t'_2. Если получился пустой унификатор, значит, эти термы равны и ограничение нарушено. В таком случае суперкомпилятор покинет эту ветвь вычислений. Если же термы не унифицируются, значит, никакая подстановка в дальнейшем не нарушит ограничение. Иначе необходимо запомнить унификатор в хранилище.

– При встрече операции унификации $t_1 \equiv t_2$ необходимо получить их унификатор. Если его не существует или он пуст, то дополнительных действий производить не нужно. Иначе нужно проверить, не нарушает ли унификатор ограничения неэквивалентности.

Указанное расширение было добавлено в библиотеку с реализацией сопуствующих алгоритмов.

Выявление остаточной программы по дереву процессов — *резидуализация* — породит новые опеределения отношений. Больше одного отношения из дерева процессов может появиться в случае, когда узлы `Renaming` указывают на узлы, отличные от корня. Поэтому первой фазой происходит пометка узлов, задающих таким образом отношения, а также удаление поддеревьев, у которых все ветви вычисления пришли к неудаче.

Далее происходит обход дерева, во время которого генерируются узлы синтаксического дерева программы в зависимости от типа текущего узла дерева процессов:

– `Unfoldable` узел приводит к появлению дизъюнкций подпрограмм, которые задают дети этого узла. Это обусловлено тем, что при прогонке в этом узле происходит ветвление вычислений;

– `Abstraction` узел приводит к появлению конъюнкций подпрограмм, которые задают дети этого узла. Это обусловлено тем, что хотя операция обобщения выявляет подконъюнкции из конфигурации и рассматривает их отдельно, оба поддерева, задающиеся этими подконъюнкциями, должны выполнятся в одно и то же время;

–

3 Тестирование

В качестве основной конкретной реализации μKanren для тестирования использовался OCanren[3][13].

3.1 Тестовое окружение

Тесты запускались на **TODO: моём ноутбуке**.

Для тестирования суперкомпилятора и его модификаций использовался следующий алгоритм.

1. На вход предоставляется программа на внутреннем DSL μKanren библиотеки.
2. Программа и запрос, на который будет происходит специализация, подаются на вход суперкомпилятору.
3. По дереву процессов, порождённому суперкомпилятором, строится остаточная программа.
4. Остаточная программа транслируется в OCanren и запускается в заранее подготовленном окружении.

[3] https://github.com/JetBrains-Research/OCanren