

# 1 Описание предметной области

## 1.1 Логическое и реляционное программирование

**Логическое программирование** — это вид декларативного программирования, основанный на формальной логике. Программы представляются в виде утверждений, представляющихся логическими формулами и описывающих определённую область проблем. “Вычисление” программы в контексте логического программирования производится в форме *поиска* доказательства утверждений на основе заданных фактов — аксиом — и правил вывода в соответствии с заданной *стратегией поиска* [39].

Стратегия поиска задаёт, каким образом происходит обход пространства поиска ответов, и, как следствие, определяет как, какие и в каком порядке будут находиться ответы. Стратегия поиска, при которой каждый возможный ответ будет со временем выдан, называется *полной*. Чаще применяются *неполные* стратегии, поскольку они менее требовательны к ресурсам [11].

Самые известные языки логического программирования — языки семейства Prolog. Prolog применяется для доказательства теорем [13], проектирования баз знаний, создания экспертных систем [29] и искусственного интеллекта [3]. Prolog строится на логике предикатов первого порядка в форме дизъюнктов Хорна (то есть дизъюнктов только с одним положительным литералом) и использует *метод резолюций*, основанный на доказательстве от противного, для решения задач. Prolog вводит разнообразные синтаксические конструкции с побочными *эффектами*, то есть с действиями, приводящими к изменению *окружения* программы, к примеру, оператор отсечения (англ. *cut*), который влияет на способ вычисления программы, предотвращая нежелательные вычисления. Также он использует стратегию обхода в глубину, что приводит к тому, что поиск может “зациклиться” и никогда не выдать оставшиеся решения, но, тем не менее, благодаря своим расширениям Prolog пригоден для решения ряда задач [39].

**Реляционное программирование** — это форма чистого логического программирования, в котором программы задаются как набор математических *отношений*. Реляционное программирование направлено на получение *значимых* ответов, как бы ни использовались отношения [4].

Исторически, понятие реляционного программирования появилось

раньше [28] и задавало саму концепцию программы как отношения, когда логическое — появилось позже и предоставляло реализацию его идей [39]. Однако в данной работе под реляционным программированием понимается непосредственно так, как указано выше.

В терминах реляционного программирования, к примеру, сложение  $X + Y = Z$  может быть выражено отношением<sup>1</sup>

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

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

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

Чистые 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 отношения вырождаются в функциональные и появляется явная зависимость 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  $\text{less}^o(X, Y)$  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.

Одно из применений реляционной парадигмы — *реляционные интерпретаторы*. Для языка  $L$  его интерпретатор — это функция  $\text{eval}_L$ , которая принимает на вход программу  $p_L$  на этом языке, её вход  $i$  и возвращает некоторый выход  $o$ :

$$\text{eval}_L(p_L, i) \equiv \llbracket p_L \rrbracket(i) = o$$

---

<sup>1</sup>Символ  $^o$  традиционно используется для обозначения отношения.

Реляционную версию интерпретатора можно представить как отношение:

$$\text{eval}_L^o(p_L, i, o).$$

При запуске такого отношения в разных направлениях можно добиться интересных эффектов: не только вычислять результат, но и по программе  $p_L$  и выходу  $o$  искать возможные входы  $i$  или вовсе генерировать программу по указанным выходам и входам.

Можно отметить, что языках, основанных на классическом Prolog, производить подобные вычисления для получения вразумительных результатов не получится.

### 1.1.1 miniKanren

*miniKanren* – семейство встраиваемых предметно-ориентированных языков, специально спроектированное для реляционного программирования[4].

Основная реализация *miniKanren* написана на языке Scheme[8], однако существует множество встраиваний в ряд других языков, в том числе Clojure, OCaml, Haskell и другие<sup>2</sup>.

*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.

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 тер-

---

<sup>2</sup>[minikanren.org](http://minikanren.org)

```

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

```

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

мы, 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  $\_0$  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 является полным [18]. 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[30], на данном этапе своего развития используется в основном в исследовательских целях:

- реляционные интерпретаторы на miniKanren для решения задач поиска [27], техника программирования по примерам [5];
- для доказательства теорем [31];
- в области вычислительной лингвистики [38].

Однако miniKanren обладает рядом существенных недостатков. Несмотря на то, что он ближе всего подошёл к реализации чистой реляционности, вычислительно он всё же зависим от сложности и ветвистости программ, из-за чего их запуск в разных направлениях может работать с разной скоростью, в особенности запуск в обратном направлении, который зачастую работает очень медленно. Также, описывать сложные задачи в качестве отношений — нетривиальная задача, наивно написанные реляционные программы вычисляются крайне неэффективно. Из-за чего, к примеру, существует транслятор из функционального языка в miniKanren [26], однако порождаемые им отношения — функциональные, а запуск их в обратном направлении, как указывалось выше, крайне непроизводителен [27].

Одно из возможных решений проблем производительности — специализация.

## 1.2 Специализация программ

**Специализация** — это метод автоматической оптимизации программ, при которой из программы удаляются избыточные вычисления, возможно, на основе информации о входных аргументах программы [16].

*Специализатор*  $\text{spec}_L$  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).

$$\llbracket \text{spec}_L(p_L, i_s) \rrbracket(i_d) \equiv \hat{p}_L(i_d) \equiv \llbracket p_L \rrbracket(i_s, i_d) \quad (1)$$

Специализатор производит все вычисления, зависящие от статических данных, протягивание констант, инлайнинг и другие.

Одно из интересных теоретических применений специализации — это *проекция Футамуры* [9]. Процесс специализации интерпретатора на про-

грамму на языке  $L$   $\text{spec}_L(\text{eval}_L, p_L)$  порождает *скомпилированную* 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. Однако реализация специализаторов, которые бы не оставляли в порождаемой программе следы интерпретации, сложная и труднодостижимая задача [16].

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

- *offline*-специализаторы — это двухфазовые алгоритмы специализации, в первой фазе которого происходит разметка исходного кода, к примеру, с помощью анализа времени связывания [16], и во второй фазе — непосредственно во время специализации — *только* на основе полученной разметки принимаются решения об оптимизации;
- *online*-специализаторы, напротив, принимают решения о специализации на лету и могут произвести вычисления, для которых *offline* сгенерировал бы код.

**TODO: Связывающие слова.**

### 1.2.1 Специализация логических языков

**Частичная дедукция** — класс методов специализации логических языков, основанных на построении деревьев вывода, которые отражают процесс вывода методом резолюций, и анализе отдельно взятых атомов логических формул [22].

Реализации методов частичной дедукции успешно применяются для Prolog [17], в частности, система *offline* частичной дедукции LOGEN показывает хорошие результаты при специализации интерпретаторов и для некоторых интерпретаторов достигает для генерируемых программ отсутствие накладных расходов на интерпретацию, однако требует ручной модификации разметки [25].

Конъюнктивная частичная дедукция — одно из расширений метода частичной дедукции, отличительная особенность которой состоит в том, что конъюнкции рассматриваются как единая сущность наравне с атомами [6].

С помощью конъюнктивной частичной дедукции возможно добиться различных оптимизационных эффектов, среди которых выделяется дефорестация и таплинг. Это наиболее проработанный и мощный метод частичной дедукции.

Реализация методов частичной дедукции, включая конъюнктивную частичную дедукцию, для Prolog представлена в виде системы ECCE [24].

В работе [27] представляется адаптация конъюнктивной частичной дедукции для miniKanren. Реализация добивается существенного роста производительности, однако, как будет показано в разделе ??, в силу особенностей метода и его направленности на Prolog, нестабильно даёт хорошие результаты и в некоторых случаях может затормозить исполнение программы.

### 1.2.2 Методы суперкомпиляции

**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. Впервые метод был предложен в работе [37].

Упрощение достигается 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 [34]. Также суперкомпиляция может применяться для специализации программ.

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*) [36]. 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*) [33].

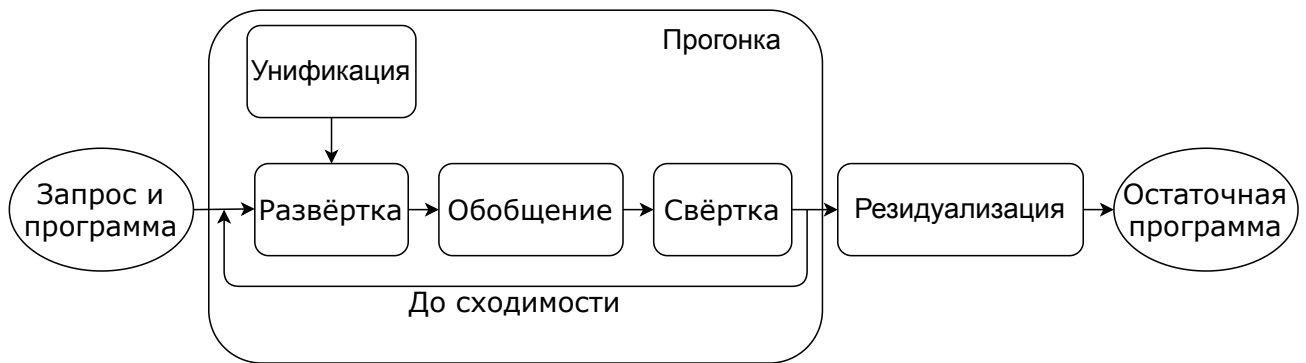


Рисунок 3 — Общая схема суперкомпилятора.

Техника суперкомпиляции в основном применяется для функциональных [2, 36] и императивных [19] языков.

Для логических языков суперкомпиляция слабо развита, однако существует ей посвящённые работы: в работе [10] демонстрируется схожесть подходов частичной дедукции и суперкомпиляции, в работе [7] представлен позитивный суперкомпилятор APROPOS для Prolog, однако эта версия довольно ограничена в своих возможностях и требует ручного вмешательства.

### 1.3 Постановка задачи

Таким образом, целью данной работы является улучшение качества специализации программ на miniKanren с использованием методов суперкомпиляции. **TODO: more.**

## 2 Имеющиеся наработки

В этом разделе описывается более подробно метод суперкомпиляции, а также библиотека для специализации miniKanren, на основе которой велась разработка.

### 2.1 Алгоритмы суперкомпиляции

Общая схема суперкомпилятора представлена на рисунке 3

История вычислений при суперкомпиляции представляется в виде *графа процессов* — корневого ориентированного графа, в котором каждая ветвь — 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[37].

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 появляется 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 выражения 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 4. Здесь при исполне-

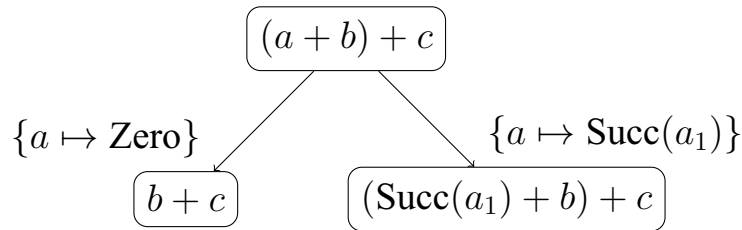


Рисунок 4 — Пример части графа процессов.

нии 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, добавлены 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 бесконечного дерева вычисления

в конечный объект, по которому можно 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 5, на котором свёрточное ребро изображено пунктиром.

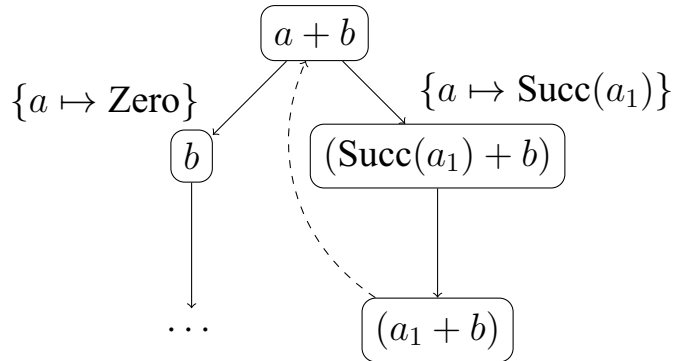


Рисунок 5 — Пример свёртки.

Однако существует ситуации, при котором свёртка не приведёт 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*[35]. Обобщение — это процесс замены одной конфигурации на другую, более абстрактную, описывающую больше состояний программы. Для обнаружения “похожей” конфигурации используется предикат, традиционно называемый *свистком*: свисток пробегает по всем родителям текущей конфигурации и определяет, похожа ли конфигурация на кого-то из них. В случае, когда свисток сигнализирует о найденной схожести, применяется обобщение. Сам шаг обобщения может произвести действия двух видов:

- *обобщение вниз* приводит к тому, что новая конфигурация заменяет текущую в графе процессов;
- *разделение* (англ. *split*) используется для декомпозиции выражений, элементы которого затем будут обработаны отдельно.

Пример обобщения представлен на рисунке 6

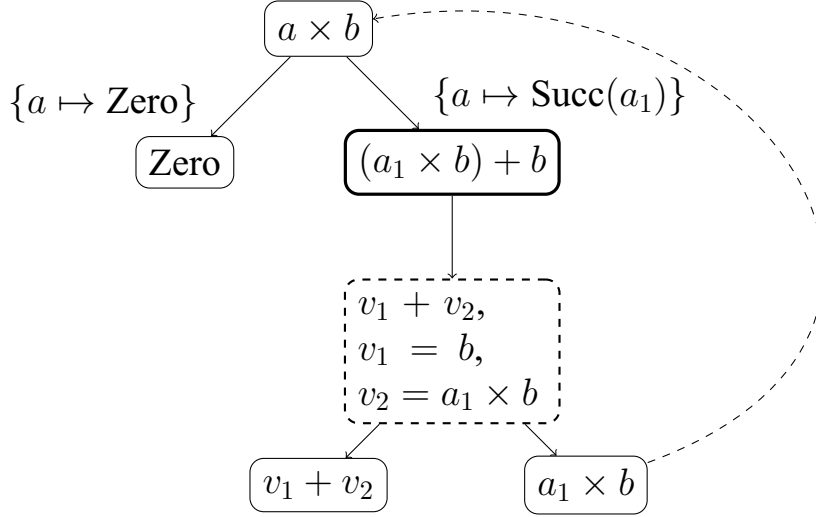


Рисунок 6 — Пример обобщения.

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.

## 2.2 Язык $\mu$ Kanren

В данной работе для специализации был выбран  $\mu$ Kanren — минималистичный диалект языка miniKanren[12].  $\mu$ Kanren содержит только чистые операторы, что значительно упрощает процесс специализации.

Абстрактный синтаксис языка представлен на Рисунке 7.

- Унификация двух термов  $t_1 \equiv t_2$  порождает подстановку  $\theta$ , 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

$\mathcal{C}$	$= \{C_i\}$	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
	$\text{fresh } \mathcal{X} . \mathcal{G}$	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  $\mu\text{Kanren}$  [32].

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  $\text{fresh } x . g$  в языках  $\text{miniKanren}$  нужно указывать явно, в отличие, к примеру, от  $\text{Prolog}$ , 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 представляются строковыми литералами, и *семантическими* переменными, которые непосредственно используются в процессе вычислений и представляются целыми числами, с которыми легче работать и генерировать свежие.

$\mu\text{Kanren}$  является ядром языка  $\text{miniKanren}$  и может быть без труда расширен необходимыми конструкциями.

## 2.3 Библиотека для специализации $\text{miniKanren}$

Реализация суперкомпилятора для  $\mu\text{Kanren}$  строилась на основе проекта по специализации  $\mu\text{Kanren}$  с помощью конъюнктивной частичной дедукции<sup>3</sup>

<sup>3</sup>[https://github.com/kajigor/uKanren\\_transformations/](https://github.com/kajigor/uKanren_transformations/)

на функциональном языке программирования Haskell. Результаты специализации  $\mu$ Kanren представлены в работе [27].

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[10], из чего следует, что ряд вспомогательных функций и определений и для частичной дедукции, и для суперкомпиляции будут совпадать;
- алгоритм обобщения конъюнктивной частичной дедукции[6], о котором подробнее будет рассказано позже, имеет ряд общих черт с процессом 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 к увеличению производительности программы, поскольку унификация — операция дорогая.

В библиотеке введены структуры данных, описывающие термы и выражения языка в соответствии с синтаксисом  $\mu$ Kanren 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.

Во-первых, в библиотеке реализован алгоритм унификации двух термов. Операция унификации находит наиболее общий унификатор (англ. *most general unifier*), причём единственный, то есть такой унификатор  $\theta$ , что для любого другого унификатора  $\theta'$  существует подстановка  $\sigma$ , с которой композиция наиболее общего унификатора даёт  $\theta'$ :  $\theta' = \sigma \circ \theta$ [21]. 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  $e_1$  ( $e_1 \preceq e_2$ ) если существует такая подстановка  $\theta$ , применение которой приравнивает два выражения  $e_1\theta = e_2$ ; также говорят, что  $e_1$  более общий, чем  $e_2$ . К примеру,  $f(X, Y) \preceq f(Y, X)$  и  $f(X, Y) \preceq f(Y, X)$ .
- Выражение  $e_2$  является *строгим* экземпляром выражения  $e_1$  ( $e_1 \prec e_2$ ), если  $e_1 \preceq e_2$  и  $e_2 \not\preceq e_1$ . 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 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 [36].

В-третьих, в качестве свистка используется отношение *гомеоморфного вложения*[35]. Отношение гомеоморфного вложения  $\sqsubseteq$  определено индуктивно:

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

$$X \sqsubseteq C_n(Y_1, \dots, Y_n) : \exists i, X \sqsubseteq Y_i;$$

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

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

К примеру, выражение  $c(b) \sqsubseteq c(f(b))$ , но  $f(c(b)) \not\sqsubseteq c(f(b))$ .

Преимущество использования гомеоморфного вложения, в первую очередь, состоит в том, что для этого отношения доказано, что на бесконечной последовательности выражений  $e_0, e_1, \dots, e_n$  обязательно найдутся такие два

индекса  $i < j$ , что  $e_i \leq e_j$ , вне зависимости от того, каким образом последовательность выражений была получена [36]. Это свойство позволяет доказать завершаемость алгоритма суперкомпиляции.

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

Отношение *строгого* гомеоморфного вложения  $\leq^+$  вводит дополнительное требование, чтобы терм  $X$ , состоящий в отношении с  $Y$ , не был *строгим экземпляром*  $Y$  [23]. В таком случае отношение  $f(X, X) \not\leq^+ f(X, Y)$ , 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 вложения было расширено на конъюнкции выражений. Пусть  $Q = A_1 \wedge \dots \wedge A_n$  и  $Q'$  — конъюнкции термов, тогда  $Q \leq Q'$ , тогда и только тогда, когда  $Q' \not\leq Q$  и существует упорядоченные подконъюнкции  $A'_1 \wedge \dots \wedge A'_n$  конъюнкции  $Q'$  (то есть  $Q'$  может содержать больше выражений, чем  $Q$ ), такие что  $A_i \leq A'_i$  [6]. Конъюнкция  $Q'$  может содержать в себе больше выражений за счёт того, что в этом случае при обобщении произойдёт 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  $e_g \preceq e_1$  и  $e_g \preceq e_2$ . 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  $e'_g \preceq e_1$  и  $e'_g \preceq e_2$  выполняется  $e'_g \preceq e_g$  [36]. 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  $\theta_1$  и  $\theta_2$  назовём *обобщающими унификаторами* (англ. *generalizers*).

Алгоритм обобщения для конъюнктивной частичной дедукции согласован с определением гомеоморфного вложения и используется в рамках конъю-

юнктивной частичной дедукции для построения более сложных алгоритмов обобщения, 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.

## 2.4 Обобщённый алгоритм суперкомпиляции

На основе вышесказанного можно построить обобщённый алгоритм суперкомпиляции, который в псевдокоде представлен на рисунке 8.

```

1  supercomp(program, query):
2    env ← createEnv program
3    configuration ← initialize query
4    graph ← emptyTree
5    drive(env, graph, configuration)
6    return residualize graph
7
8  drive(env, graph, configuration):
9    if configuration is empty
10   then add(env, graph, success node)
11   else if  $\exists$  parent: configuration  $\approx$  parent
12   then add(env, graph, renaming node)
13   else if  $\exists$  parent: parent  $\leq^+$  configuration
14   then
15     add(env, graph, abstraction node)
16     children ← generalize(configuration, parent)
17      $\forall$  child  $\in$  children:
18       drive(env, graph, child)
19   else
20     add(env, graph, unfolding node)
21     children ← unfold(env, configuration)
22      $\forall$  child  $\in$  children:
23       drive(env, graph, child)

```

Рисунок 8 — Обобщённый алгоритм суперкомпиляции.

Алгоритм суперкомпиляции принимает на себя программу и запрос, на который необходимо специализировать программу, и после инициализации начальных значений, включающих в себя некоторое *окружения программы*



(`env` на строке 8), в котором хранятся все вспомогательные структуры, запускает процесс прогонки. Прогонка производится до схождения и производит следующие действия в зависимости от состояния:

- если конфигурация пустая (строка 9), это означает, что вычисления успешно сошлись в конкретную подстановку. В таком случае происходит добавление в граф листового узла с этой подстановкой;
- если существует такая родительская конфигурация, что она является вариантом текущей (строка 11), то происходит свёртка и в граф добавляется листовой узел с ссылкой на родителя;
- если же среди родителей находится такой, на котором срабатывает свисток (строка 13), тогда производится обобщение, порождающее дочерние конфигурации (строка 16), на которых продолжается процесс прогонки;
- иначе происходит шаг символьного вычисления (строка 19), на котором порождаются конфигурации.

После завершения этапа прогонки из графа процессов извлекается остаточная программа (строка 6)

## 3 Суперкомпиляция `miniKanren`

### 3.1 Реализация суперкомпилятора

Представление графа процессов в Haskell затруднено тем, что графовые структуры данных обычно требуют ссылок на произвольные узлы, что приводит к появлению перекрёстных ссылок. Прямая реализация этой идеи сложна в разработке и поддержке и не является идиоматичной. Использование `IORef`<sup>4</sup>, хотя и предоставляет мутабельность, приводит к неоправданному усложнению кода всего проекта, лишая код 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

<sup>4</sup><https://hackage.haskell.org/package/base-4.11.1.0/docs/Data-IORef.html>

представить граф в качестве *дерева* процессов. Древоподобная структура однозначно отображается на процесс символьных вычислений, а также с ними легко и идиоматично работать в Haskell.

Структура дерева процессов представлена на рисунке 9.

```

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

```

Рисунок 9 — 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 отображение 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` обозначает узел, который может быть обобщён на одного из родителей; После обобщения может появиться несколько конфигураций, 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` обозначает шаг символьного вычисления, на котором произошёл шаг вычислений и по рассматриваемой на этом шаге конфигурации породились новые конфигурации.

Базовой стратегией порождения новых конфигураций выбрана *полная стратегия развёртывания*, пример которой был представлен в предыдущем разделе, при которой мы заменяем определения всех реляционных вызовов 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 его тело при развёртывании.

## 3.2 Модификации суперкомпилятора

### **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 часто встречается, что в разных поддеревьях дерева процессов встречаются одинаковые конфигурации, поддеревья которых оказываются полностью идентичными. В таком случае, кажется очевидной оптимизация, при которой мы запоминаем вычисленные поддеревья и в случае, когда мы встречаем схожую конфигурацию, не вычисляем поддерево заново, добавляя ссылку на него. **TODO: Вопрос на засыпку меня же: может ли быть такое, что конфигурации являются вариантами друг друга, но накопленные подстановки приведут к появлению различных поддеревьев?**

**Обобщение на все вычисленные узлы, не только на родительские**

**TODO: Понять, почему это не противоречит методам суперкомпиляции**

Обобщение на вычисленные узлы приводит к тому, что деревья конфигураций быстрее сходятся, однако остаётся под вопросом, ухудшает ли этот подход качество специализации.

### **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 *таплингу* (англ.  *tupling* )[14] — оптимизации, 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 произведение тел вызовов в конъюнкциях, что приводит к сильному разрастанию дерева процессов и, как следствие, 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 *Unfoldable* (рисунок 10).

Предоставляемые интерфейсом функции используются в алгоритме суперкомпиляции следующим образом:

```

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

```

Рисунок 10 — Интерфейс для различных стратегий развёртывания.

- `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.
- **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 `ando`) или привести к скорой свёртке.

- **Рекурсивная стратегия развёртывания** при которой в первую очередь раскрывается рекурсивный вызов в конфигурации. **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.

- **Стратегия развёртывания вызовов с максимальным количеством ветвлений**, при которой на каждом шаге вычисления будет появляться максимально возможное количество конфигураций, что, с одной стороны, увеличит количество возможных состояний, но потенциально может привести к сворачиванию или обобщению.

### Реализация обобщения вверх

В суперкомпиляции, в отличие от методов частичной дедукции, для обобщения дополнительно может использоваться техника обобщения вверх, при которой происходит не подвешивание обобщённой конфигурации в качестве потомка конфигурации, которая обобщалась, но замена самого родителя на новую конфигурацию. Старое же поддереву родителя уничтожается.

Для определения необходимости обобщать вверх введём предикат  $e_1 \triangleleft e_2$ , 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**

```

1  else if  $\exists$  parent: parent  $\triangleleft$  configuration
2  then
3      node  $\leftarrow$  generalize(configuration, parent)
4      addUp(env, tree, parent, node)

```

Рисунок 11 — Расширение алгоритма 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 *zipперы*[15]. 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 12).

```

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

```

Рисунок 12 — 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: Описать методику передвижения.**

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

Обобщение вверх приводит к тому, что происходит замена целого поддерева процессов предка, на которого обобщается конфигурация. Иногда это может приводить к потере связи между аргументами, из-за чего исчезает потенциал для возможных потоложительных эффектов, к примеру, протягивания констант.

К примеру, на рисунке 13 представлено дерево процессов, при котором происходит обобщение вверх.

С одной стороны, теряется потенциал для генерации более оптимальной для цели  $\text{reverse}^o(a, a)$  программы, но с другой стороны рассмотрим следующие соображения.

В данном примере процесс прогонки происходил следующим образом:

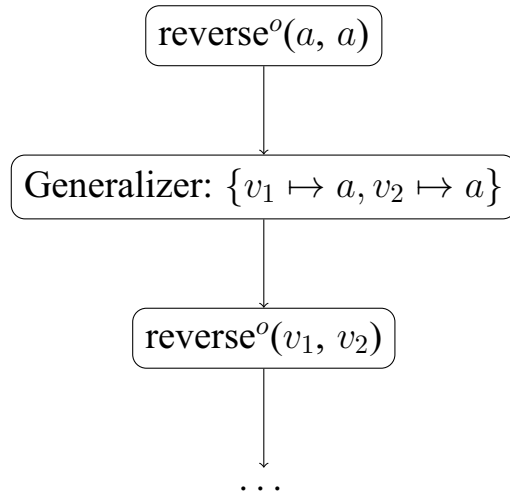


Рисунок 13 — Демонстрация потери информации при обобщении вверх.

некоторое время строился граф для конфигурации  $\text{reverse}^o(a, a)$ , затем вывелась конфигурация  $\text{reverse}^o(a, b)$ . Если бы обобщения вверх не происходило бы, то поиск ответов в результирующей программе малое время провёл бы в поддереве, оптимизированном под  $\text{reverse}^o(a, a)$ , а остальное — в обобщённом  $\text{reverse}^o(a, b)$ .

В общем случае это может происходить не только с корнем дерева, но и в каких-то его поддеревьях. Однако запрет на обобщение вверх в поддеревьях может сгенерировать слишком много частных случаев и привести к более неэффективным программам.

Из того, что, во-первых, есть потенциал оптимизации при сохранении информации в корне дерева и, во-вторых, необходимо сдерживать разрастание дерева конфигураций, допускается рассмотрение алгоритма с обобщением вверх с запретом на обобщение к корню дерева.

### Расширение языка $\mu\text{Kanren}$ с помощью операции неэквивалентности

Множество операции в оригинальном  $\mu\text{Kanren}$  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-значение  $\text{lookup}^o$ .

В соответствии с программой список  $L$  должен иметь в голове пару



```

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.

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  $(K', V) :: L' \equiv L$  прошла успешно и был найден результат, то поиск всё равно пройдёт 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  $t_1 \not\equiv t_2$ , вводящее ограничение неэквивалентности (англ. *disequality constraints*)[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  $\mu$ Kanren представлено на рисунке 15.

$$\mathcal{G} = \dots \quad \mathcal{T}_x \not\equiv \mathcal{T}_x \quad \text{дезунификация}$$

Рисунок 15 — Расширение синтаксиса  $\mu$ Kanren относительно указанного на рисунке 7.

Исправленная версия отношения  $\text{lookup}^o$  представлена на рисунке 16.

```

1 lookupo K L R =
2   (K', V) :: L' ≡ L ∧
3   (K' ≡ K ∧ V ≡ R ∨
4   K' \not\equiv K ∧ lookupo 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”  $\Omega$  (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 сопствующих 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, ко-

торые 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 выполняться в одно и то же время;
- `Generalizer` задаёт обобщающий унификатор, который должен быть добавлен перед своим поддеревом;
- `Renaming` формирует вызов реляционного отношения;
- `Success` представляет собой успешное вычисление, предоставляющее непротиворечивую подстановку.

## 4 Тестирование

### 4.1 Тестовое окружение

В качестве основной конкретной реализации  $\mu$ Kanren для тестирования использовался OCanren<sup>5</sup>[20], встроенный в OCaml[20]. Для некоторых тестов для использовался faster-miniKanren<sup>6</sup>, версия miniKanren, встроенная в Scheme.

Тесты запускались на платформе: Intel Core i5-6200U CPU, 2.30GHz, DDR4, 12GiB.

Для тестирования суперкомпилятора и его модификаций использовался следующий алгоритм.

1. На вход предоставляется программа на внутреннем DSL  $\mu$ Kanren библиотеки специализации.
2. Программа и запрос, на который будет происходить специализация, подаются на вход суперкомпилятору.
3. По дереву процессов, порождённому суперкомпилятором, строится остаточная программа.

---

<sup>5</sup><https://github.com/JetBrains-Research/OCaml>

<sup>6</sup><https://github.com/miniKanren/faster-miniKanren>

4. Остаточная программа транслируется в OCanren/faster-miniKanren и запускается в заранее подготовленном окружении с тестовыми запросами.

Реализованный суперкомпилятор сравнивался с реализацией конъюнктивной частичной дедукции для  $\mu$ Kanren<sup>7</sup>, а также с реализацией конъюнктивной частичной дедукции для Prolog — системой ECCE<sup>8</sup>. Для последнего требовалось оттранслировать программу на  $\mu$ Kanren в Prolog, специализировать её на запрос, далее оттранслировать результирующую программу в OCanren. Все необходимые средства для этого также предоставлялись указанной библиотекой специализации.

#### 4.1.1 Набор тестов

- Отношение сортировки  $\text{sort}^o(\text{list}, \text{result})$ . Запросы:
  - оптимизация сортировки:  $\text{sort}^o(\text{xs}, \text{ys})$ ;
  - генерация отсортированных последовательностей:  $\text{sort}^o(\text{xs}, \text{xs})$ .
- Отношение, проверяющее принадлежность пути графу  $\text{isPath}^o(\text{path}, \text{graph}, \text{result})$ . Специализация  $\text{isPath}^o(\text{path}, \text{graph}, \text{true})$ . Запросы:
  - генерация  $n$  произвольных путей в случайном графе;
  - поиск пути заданного размера в случайном графе:  
 $\text{isPath}_s^o(p, g) \wedge \text{length}^o(p, N)$ .
- Интерпретатор формул логики высказываний  $\text{logint}^o(\text{formula}, \text{subst}, \text{result})$ . Запросы:
  - поиск  $n$  решений заданной формулы;
  - генерация  $n$  формул с подстановке размера  $n$ .
- Интерпретатор лямбда-исчисления  $\text{lam}^o(\text{expr}, \text{result})$ . Запросы:
  - генерация  $n$  выражений в нормальной форме  $\text{lam}^o(\text{expr}, \text{expr})$ ;
  - генерация  $n$  выражений, редуцирующихся к заданному выражению  $\text{lam}^o(\text{expr}, E)$ .

<sup>7</sup>[https://github.com/kajigor/uKanren\\_transformations](https://github.com/kajigor/uKanren_transformations)

<sup>8</sup><https://github.com/leuschel/ecce>

- Проверка типов в просто типизированном лямбда-исчислении  $\text{infer}(\text{type}, \text{expr})^o$ .
  - Поиск  $n$  обитателей заданного типа.
  - Генерация выражений, соответствующих заданной спецификации типа и выражения. Специализируется выражение:
 
$$\text{infer}^o(\text{type}, \text{expr}) \wedge \text{type} \equiv \text{TYPE\_SPEC} \wedge \text{expr} \equiv \text{EXPR\_SPEC}.$$
- Интерпретатор простого подмножества Scheme.
  - **TODO: Интересный тест!**