ВВЕДЕНИЕ

Реляционное программирование — это чистая форма логического программирования, в которой программы представляются как наборы математических отношений [4]. Отношения не делают различие между выходными и выходными параметрами, из-за чего одно и то же отношение может покрыть несколько связанных проблем. К примеру, отношение, задающее интерпретатор языка, можно использовать не только для вычисления программ по заданному входу, но и генерировать возможные входные значения по заданному результату или генерировать сами программы по спецификации входных и выходных значений интерпратетора.

miniKanren — это семейство встраиваемых предметно-ориентированных языков программирования [4]. miniKanren был специально сконструирован для того, чтобы поддержать реляционную парадигму, опираясь на опыт логических языков, таких как языков семейства Prolog [46], Mercury [38] или Curry [11].

Однако реляционная парадигма довольно сложна, хотя потенциал её весьма велик. Часто наиболее простой способ записи отношения не является эффективным. В частности, при задании функциональных отношений как сопоставления выходов входам, как это наблюдается в примере с интерпратетором, практически всегда работает медленно.

Специализация – это техника автоматической оптимизации программ, при которой на основе программы и её возможно частично известного входа порождается новая, более оптимальная программа, которая сохраняет семантику исходной. Для специализации логических языков используются методы частичной дедукции [24], самый проработанный из которых — это конъюнктивная частичная дедукция [6]. Реализация конъюнктивная частичная дедукция для Prolog ECCE показывает хорошие результаты [27], однако специфика реляционного программирования и отличие его от логических языков подразумевает возможность разработать более подходящий метод специализации. Уже существует адаптация конъюнктивная частичная дедукция для miniKanren [30], однако её результаты нестабильны, и несмотря на то, что в некоторых случаях производительность программ улучшается, в других – она может существенно ухудшиться.

Другой подход для специализации — это суперкомпиляция, техни-

ка автоматической трансформации и анализа программ, при которой программа символьно исполняется с сохранением истории вычислений, на основе которой принимаются решения о трансформациях. Суперкомпиляция успешно применяется к функциональным и императивным языкам, однако суперкомпиляция для логическим языкам не сильно развита, хотя и существуют работы, посвящённые демонстрации сходства процессов частичной дедукции и суперкомпиляции [10], а также суперкомпилятор APROPOS [7], который, однако, довольно ограничен в своих возможностях и требует ручного контроля.

В данной работе предлагается способ адаптации и реализации суперкомпилятора для реляционного языка miniKanren, также рассматриваются его возможные вариации для повышения производительности реляционных программ и производится апробация результата.

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* [46].

Стратегия поиска задаёт, каким образом происходит обход пространства поиска ответов, и, как следствие, определяет как, какие и в каком порядке будут находится ответы. Стратегия поиска, при которой каждый возможный ответ будет со временем выдан, называется *полной*. Чаще применяются *неполные* стратегии, поскольку они менее требовательны к вычислительным ресурсам [12].

Самые известные языки логического программирования — языки семейства Prolog. Prolog применяется для доказательства теорем [14], проектирования баз знаний, создания экспертных систем [32] и искусственно-

го 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 остаётся хорошим решением для задач из своей области применения [46].

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 [31] и задавало саму концепцию программы как отношения, когда логическое — появилось позже и предоставляло реализацию его идей [46]. 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[1]

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

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.

[1] Символ ^{o} традиционно используется для обозначения отношения.

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 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.

Одно из применений реляционной парадигмы — *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:

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

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 реализаций в ряде других языков, в том числе Clojure, OCaml, Haskell и другие[2].

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 — пе-

[2] `minikanren.org`

ременные, 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 [19]. 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 [33], на данном этапе своего развития используется в основном в исследовательских целях:

- реляционные интерпретаторы на miniKanren для решения задач поиска [30], техника программирования по примерам [5];
- для доказательство теорем [34] ;
- в области вычислительной лингвистики [43].

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 [29], одна-

ко порождаемые им отношения — функциональные, а запуск их в обратном направлении, как указывалось выше, крайне непроизводителен [30].

Одно из возможных решения 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 вхоных аргументах программы [17].

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* [9]. 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. Однако реализация специализаторов, которые бы не оставляли в порождаемой программе следы интерпретации, сложная и труднодостижимая задача [17].

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 [17], 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 сгенеировал бы код.

TODO: Связывающие слова.

1.2.1 Специализация логических языков

Частичная дедукция — класс методов специализации логический языков, основанных на построении деревьев вывода, которые отражают процесс вывода методом резолюций, и анализе отдельно взятых атомов логических формул [24].

Реализации методов частичной дедукции успешно применяются для Prolog [18], 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 [28].

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 (англ. *deforestation*) [44] — оптимизация, при которой удаляются промежуточные структуры данных, — и таплинг (англ. *tupling*) [15] — оптмизация, при которой множество проходов по одной структуре данных заменяется на один проход. Это наиболее проработанный и мощный метод частичной дедукции.

Реализация методов частичной дедукции, включая конъюнктивную частичную дедукцию, для Prolog представлена в виде системы ECCE [26].

В работе [30] 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 [42].

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 [39]. 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*) [41]. 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*) [37].

Техника суперкомпиляции в основном применяется для функциональных [2, 41] и императивных [20] 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

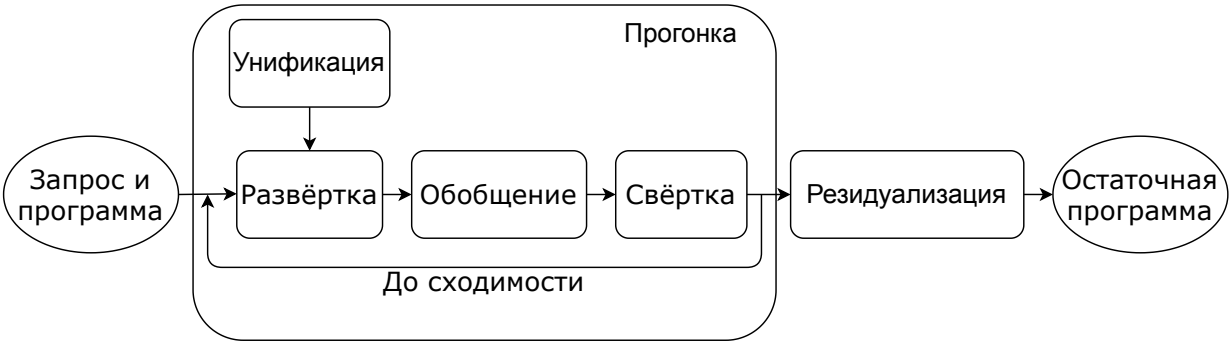

Рисунок 3 — Общая схема суперкомпилятора.

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 [42].

Прогонкой (англ. *driving*) называется процесс построения графа процессов. 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 испол-

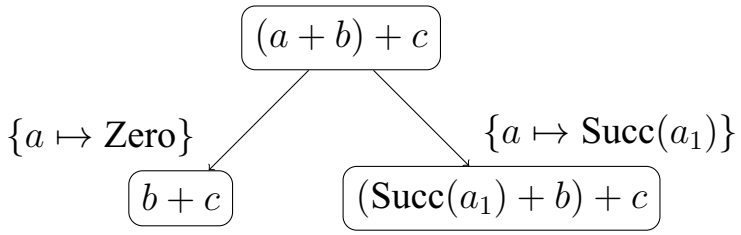

Рисунок 4 — Пример части графа процессов.

нении 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 бесконечого 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.

Однако существует ситуации, при котором свёртка не приведёт к то-

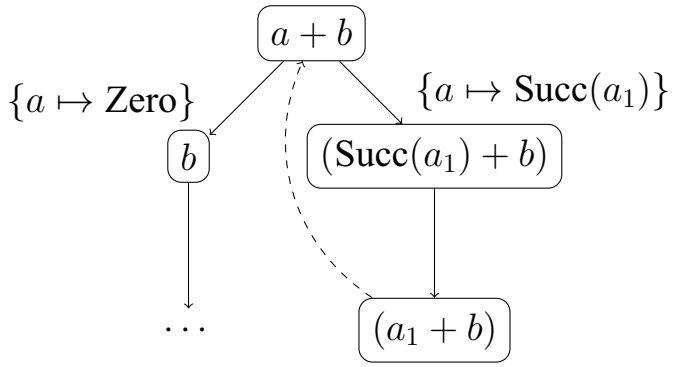

Рисунок 5 — Пример свёртки.

му, 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* [40]. 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 выражении, элементы которого затем будут обработаны отдельно.

Пример обобщения представлен на рисунке 6

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.

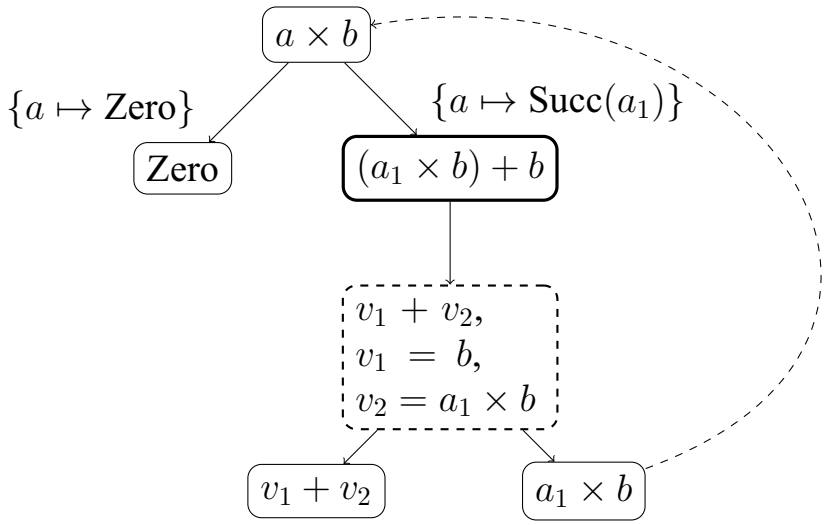

Рисунок 6 — Пример обобщения.

2.2 Язык μKanren

В данной работе для специализации был выбран μKanren — минималистичный диалект языка miniKanren [13]. μKanren содержит только чистые операторы, что значительно упрощает процесс специализации.

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
$\mathcal{D}$	$=$	$\mathcal{T}_\emptyset$	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 [36].

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

μKanren является ядром языка miniKanren и может быть без труда расширен необходимыми конструкциями.

2.3 Библиотека для специализации miniKanren

Реализация суперкомпилятора для μKanren строилась на основе проекта по специализации μKanren с помощью конъюнктивной частичной дедукции[3] на функциональном языке программирования Haskell. Результаты специализации μKanren представлены в работе [30].

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

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 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.

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 θ, что для любого другого унификатора θ' существует подстановка σ, с которой композиция наиболее общего унификатора даёт θ': $\theta' = \sigma \circ \theta$ [23]. 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 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 [41].

В-третьих, в качестве свистка используется отношение *гомеоморфного вложения* [40]. 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 [41]. Это свойство позволяет доказать завершаемость алгоритма суперкомпиляции.

Однако отношение гомеоморфного вложения допускает, чтобы термы $f(X, X)$ и $f(X, Y)$ находились в отношении $f(X, X) \trianglelefteq f(X, Y)$ 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 $\trianglelefteq^+$ вводит дополнительное требование, чтобы терм X, состоящий в отношении с Y, не был *строгим экземпляром* Y [25]. 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 $Q' \not\preccurlyeq Q$ и существует упорядоченные подконъюнкции $A'_1 \wedge \cdots \wedge A'_n$ конъюнкции Q' (то есть Q' может содержать больше выражений, чем Q), такие что $A_i \trianglelefteq A'_i$ [6]. 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 $e'_g \preccurlyeq e_g$ [41]. 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 обобщения для конъюнктивной частичной дедукции согласован с определением гомеоморфного вложения и используется в рамках конъюнктивной 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 ∃ parent: configuration ≈ parent
12   then add(env, graph, renaming node)
13   else if ∃ parent: parent ◁⁺ configuration
14   then
15     add(env, graph, abstraction node)
16     children ← generalize(configuration, parent)
17     ∀child ∈ children :
18       drive(env, graph, child)
19   else
20     add(env, graph, unfolding node)
21     children ← unfold(env, configuration)
22     ∀child ∈ 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.

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.

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

3 Суперкомпиляция miniKanren

3.1 Реализация суперкомпилятора

В качестве конкретного языка для реализации суперкомпилятора был выбран Haskell. Самая существенная причина выбора в том, на этом языка написана разобранная в предыдущем разделе библиотека для специализации языка μKanren, которую можно эффективно переиспользовать для

реализации суперкомпилятора. Несмотря на то, что некоторые техники суперкомпиляции сложно выразить в Haskell, в основном они с лёгкостью перекладываются на функциональную парадигму.

Рассмотрим более внимательно схему суперкомпилятора на рисунке 9. На нём жёлтым цветом выделены модули, которые были взяты из библиотеки специализации для построения суперкомпилятора.

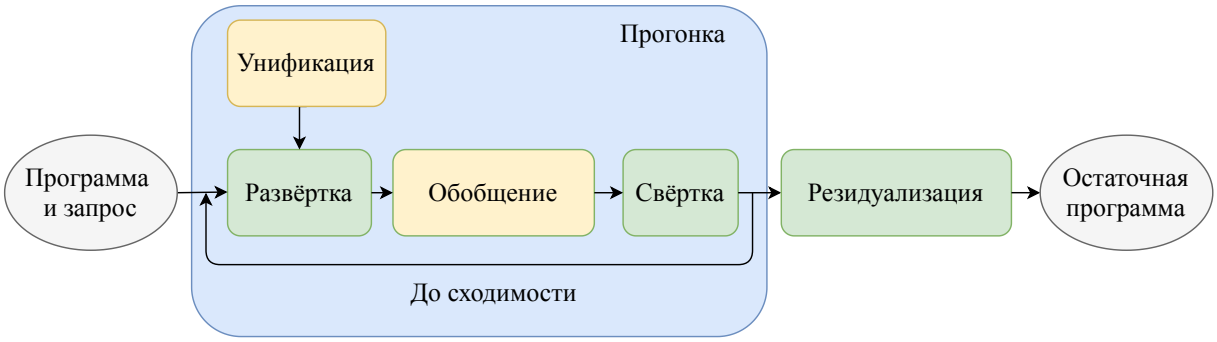

Рисунок 9 — Уточнённая схема суперкомпилятора.

Тогда необходимы модули свёртки, развёртки, резидуализации, а также модифицированный под задачу модуль обобщения, которые затем нужно собрать в единый суперкомпилятор:

– реализация модуля свёртки заключается всего лишь в стратегии выбора множества конфигураций, на которые можно применять непосредственно операцию свёртки;
– реализация модуля развёртки требует проработки стратегий символьного вычисления применительно для μKanren;
– реализация всего суперкомпилятора требует проработку структур данных, самая важная из которых — граф процессов;
– реализация резидуализации состоит из обхода графа процессов, во время которого происходит построение остаточной программы.

3.1.1 Граф процессов

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, хотя

и используется в реализациях суперкомпиляторов для небольших функциональных языков [45]. Использование *IORef*[4], хотя и предоставляет изменяемость, приводит к неоправданному усложнению кода всего проекта, лишая код функциональной чистоты. Также есть возможность использовать структуру данных отображения для представления графа процессов, как это сделано в реализации на Haskell работы [21], однако в данной работе используется метод, который обычно применяется в частичной дедукции.

Заметим, что история вычислений представляется в виде графа из наличия *обратных рёбер* — то есть рёбра от детей к их родителям, — которые появляются при свёртке, когда ребёнок является переименованием родителя. Тогда, если уметь сохранять или восстанавливать информацию об этой связи, то достаточно будет представить граф в качестве *дерева* процессов. Древовидная структура однозначно отображается на процесс символьных вычислений, а также с ними легко и идиоматично работать в Haskell.

Структура дерева процессов представлена на рисунке 10.

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

Рисунок 10 — 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

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

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

Суперкомпилятор строит дерево процессов в глубину.

3.1.2 Развёртка

Стратегия символьного вычисления определяется на этапе развёртки: происходит “развёртывание” одного или более вызовов реляционного отношения, при котором происходит замена вызова на определение. Развёртка 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 [36], 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 прямым образом.

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 11, в котором определены отношения `f` и `g`.

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

Рисунок 11 — Пример отношений для демонстрации шага символьных вычислений

Допустим, на шаге суперкомпиляции алгоритм обрабатывает конфигурацию $\texttt{f(v}_1\texttt{)}\wedge\texttt{g(v}_1\texttt{, v}_2\texttt{)}$ хотим сделать шаг символьного вычисления. Рассмотрим несколько способов породить новые конфигурации.

– Если раскроется определение `f`, то будут получены новые конфигурации $\texttt{f'(v}_1\texttt{)}\wedge\texttt{g(v}_1\texttt{, v}_2\texttt{)}$ и $\texttt{f''(v}_1\texttt{)}\wedge\texttt{g(v}_1\texttt{, v}_2\texttt{)}$.

– Если раскроется определение `g`, то будет получена новая конфигурация $\texttt{f(v}_1\texttt{)}\wedge\texttt{g'(v}_1\texttt{)}\wedge\texttt{g''(v}_2\texttt{)}$.

– Если раскроются оба определения `f` и `g`, то будут получены новые конфигурации $\texttt{f'(v}_1\texttt{)}\wedge\texttt{g(v}_1\texttt{)}\wedge\texttt{g''(v}_2\texttt{)}$ и $\texttt{f''(v}_1\texttt{)}\wedge\texttt{g(v}_1\texttt{)}\wedge\texttt{g''(v}_2\texttt{)}$.

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 по-

следующих шагах вычисления, если перед этим ветвь исполнения не будет остановлена из-за противоречивой подстановки.

Таким образом, какой бы способ развёртывания определений ни 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 — к различным эффектам специализации.

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

3.1.3 Резидуализация

Выявление остаточной программы по дереву процессов — *резидуализация* — породит новые опередеделения 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;
- `Generalizer` задаёт обобщающий унификатор, который должен быть добавлен перед своим поддеревом;
- `Renaming` формирует вызов реляционного отношения;

– `Success` представляет собой успешное вычисление, предоставляющее непротиворечивую подстановку.

3.2 Модификации суперкомпилятора

Предполагается, что базовый алгоритм специализации должен быть хуже по скорости и, в некоторых случаях, по качеству специализации. Данная работа не затрагивает способы повышения эффективности процесса суперкомпиляции, однако демострирует некоторые продвижения в эту сторону из-за чисто прагматических соображений. Основное внимание уделяется подходам к суперкомпиляции и их влиянию на суперкомпиляционные эффекты.

3.2.1 Стратегии свёртки

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 такая ситуация, что в разных поддеревьях дерева процессов стречаются одинаковые конфигурации, поддеревья которых оказываются полностью идентичными. В таком случае, кажется очевидной и несложной оптмизация, при которой мы запоминаем вычисленные поддеревья и в случае, когда мы встречаем схожу конфигурацию, не вычисляем поддерево заново, добавляя ссылку на него.

3.2.2 Стратегии развёртки

Как уже говорилось, разные стратегии развёртывания реляционных вызовов могут привести к разным эффектам специализации. К примеру, полная стратегия развёртывания, которая была принята за базовую, может приводить к дефорестации, описанному ранее.

Основной недостаток базового подхода в том, что для получения всех возможных состояний он производит декартово произведение нормализованных состояний тел вызовов в конъюнкциях, что приводит к сильному разрастанию дерева процессов и, как следствие, сильно требователен

к вычислительным ресурсам и приводит к большой ветвистости программ. Последнее оказывает негативный эффект на процесс вычисления и может ухудшить производительность. Вследствие чего реализация новых стратегий развёртывания производится в исследовательских и прикладных целях.

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

```
1 class Unfoldable a where
2    initialize :: Conf → a
3    get        :: a → Conf
4    unfold     :: 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;
- `unfold` 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, описанных ниже.

- **Модифицированная полная стратегия развёртки**, при которой сначала из цели раскрываются все нерекурсивные вызов. 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.
- **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.

  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.

– **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 максимльным 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 скорому сворачиванию или обобщению.

3.2.3 Стратегии обобщения

Для модификации стратегии обобщения были рассмотрены два подхода.

Во-первых, увеличение множества рассматриваемых конфигураций, на которые производится обобщение, до всех уже рассмотренных конфигураций.

Покажем, что допустимо использовать обобщение на все вершины. Для этого рассмотрим конфигурацию C_w и некоторую конфигурацию C_p из множества конфигураций для обобщения, расширенное всеми уже обработанными конфигурациями. При обобщении C_p и C_w породится множество дочерних конфигураций, каждая из которых будет более общим подконъюнтом C_w и, в силу своей обобщённости, будет содержать меньше информации, чем исходная конфигурация, однако не будет ей противоречить. В

итоге, обобщение на все обработанные вершины влияет только на то, каким образом разбивается рассматриваемая конфигурация, что может привести к скорой свёртке, и как много информации о переменных теряется.

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

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 [41]. Старое же поддерево родителя уничтожается.

Для определения необходимости обобщать вверх введём предикат $e_1 \preccurlyeq e_2$, 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 замена родительского дерева, во-первых, не добавит конфигураций, которых там не может быть, во-вторых, предоставит только более общую информацию.

```
1  else if ∃ parent: parent ≼ configuration
2  then
3     node ← generalize(configuration, parent)
4     addUp(env, tree, parent, node)
```

Рисунок 13 — 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 *зиппе-ры* (англ. *zipper*) [16], на основе которых, к примеру, создан система для суперкомпиляции, представленная в работе [35].

Идиома зипперов 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, 5]` при фокусе на 3 представляется таким образом: `(3, ([2, 1], [4, 5]))`. 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 14).

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

Рисунок 14 — 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. Тогда передвижение вверх до первого подходящего узла происходит путём заполнения дыры при переходе на родителя, а передвижение в левого брата — простой сдвиг в зиппере списка `children`.

При представлении дерева процессов в идиоме зипперов основа алгоритма суперкомпиляции принимает форму описания действий при смене состояния зиппера. Дерево всё ещё строится в глубину и происходит это следующим образом: если мы в узле, порождающем другие узлы (то есть `Unfolding` или `Abstraction`), то порождённые узлы добавляются в дерево с пометкой о том, что они не достроены, а алгоритм спускается в первого ребёнка. Когда же алгоритм приходит в листовой узел, то ему нужно подняться до родителя, у которого существует помеченный ребёнок, и спуститься в этого ребёнка. Алгоритм завершается, когда не осталось

помеченных детей.

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 15 представлено дерево процессов, при котором происходит обобщение вверх.

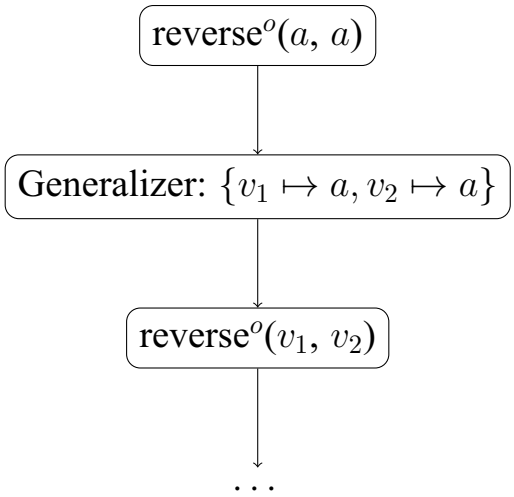

Рисунок 15 — Демонстрация потери информации при обобщении вверх.

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

В данном примере процесс прогонки происходил следующим образом: некоторое время строился граф для конфигурации reverse$^o(a,\ a)$, затем вывелась конфигурация 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 reverse$^o(a,\ a)$, а остальное — в обобщённом 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 сдреживать разрастание дерева конфигураций, допускается рассмотрение алгоритма с обобщением вверх с запретом на обобщение к корню дерева.

3.2.4 Расширение μKanren

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 16 изображена операция поиска значения по ключу в списке пар ключ-значения lookup^o.

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

Рисунок 16 — 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 17.

Исправленная версия отношения lookup^o представлена на рисунке 18.

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

Рисунок 17 — Расширение синтаксиса μKanren относительно указанного на рисунке 7.

```
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 18 — 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.

4 Апробация

4.1 Тестовое окружение

В качестве основной конкретной реализации μKanren для тестирования использовался OCanren[5][22], реализованный на OCaml[22]. Для некоторых тестов для использовался faster-miniKanren[6], версия miniKanren, реализованная на Scheme.

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

Для тестирования суперкомпилятора и его модификаций использовался следующий алгоритм.

1. Подготавливается программа, реализованная на внутреннем представлении μKanren .
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/faster-miniKanren и запускается в заранее подготовленном окружении с тестовыми запросами.

Реализованный суперкомпилятор сравнивался с реализацией конъюнктивной частичной дедукции для μKanren[7], а также с реализацией конъюнктивной частичной дедукции для Prolog — системой ECCE[8]. Другие специализаторы не рассматриваются, так как согласно работе [27], специали-

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

[6]https://github.com/miniKanren/faster-miniKanren

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

[8]`https://github.com/leuschel/ecce`

зация с помощью конъюнктивная частичная дедукция в ECCE показывает лучшие результаты.

Для последнего требовалось оттранслировать программу на μKanren в Prolog, специализировать её на запрос, далее оттранслировать результирующую программу в OCanren. Это допустимо сделать в силу того, что между μKanren и подмножеством Prolog есть взаимооднозначное соответствие. Все необходимые средства для этого также предоставлялись указанной библиотекой специализации.

4.1.1 Набор тестов

Был выбран следующий набор тестов для тестирования и анализа суперкомпилятора и его модификаций.

- Программа `doubleAppend(xs, ys, zs, rs)`, которая производит конкатенацию трёх списков. Она классически используется для проверки эффекта дефорестации в специализированной программе.
- Программа `maxLength(xs, max, len)`, которая находит в списке максимальный элемент и длину списка. Она классически используется для проверки эффекта таплинга в специализированной программе.
- Программа сортировки `sort(list, result)`. Выбрана в силу показательности результатов.
- Интерпретатор формул логики высказываний `loginto(formula, subst, result)`. Интерпретатор специализируется на то, чтобы всегда генерировать выполнимые формулы `logint(formula, subst, true)`.

Все вышеперечисленные программы были применены в реализованным суперкомпиляторами, и результаты выполнения остаточных программ были проверены на адекватность и соответствие семантике исходной программы.

4.2 Сравнение вариаций суперкомпилятора μKanren

В таблицах используются условные обозначения для стратегий развёртывания:

– *Full* и *Full-non-rec* обозначают полную стратегию и полную стратегию развёртывания с приоритетом на нерекурсивные вызовы соответственно;

– *Seq* обозначает последовательную стратегию развёртывания;

– *Non-rec* и *Rec* обозначают нерекурсивную и рекурсивную стратегии соответственно;

– *Min* и *Max* обозначают минимальную и максимальную стратегии соответственно;

– *First* обозначает стратегию, при которой всегда развёртывается первый конъюнкт.

А также для суперкомпиляторов:

– *Б.С.* обозначает базовый суперкомпилятор с обобщением вниз на предков;

– *М.1* обозначает модификацию, при которой происходит запрет на обобщение после обобщения;

– *М.2* обозначает модификацию, при которой обобщение происходит на все вычисленные вершины;

– *М.3* обозначает модификацию, при которой происходит обобщение вверх на родительские вершины;

– *М.4* обозначает модификацию, при которой происходит обобщение вверх на родительские вершины, кроме корневой.

В таблице 1 представлены результаты сравнения модификаций с базовым суперкомпилятором при разных стратегиях развёртывания. При полных стратегиях для `doubleAppendo` возникает эффект дефорестации. В остальных стратегиях этого не происходит, из-за чего исполнения по крайней мере в два раза хуже. Из-за того, что программа довольно небольшая, модификации алгоритма суперкомпиляции хотя не и оказывают влияния, результат не портят.

В таблице 2 указаны результаты тестирования для программы `maxLengtho`.

В таблице 3 указаны результаты тестирования для программы `sorto`.

	Б.С.	*М.1*	*М.2*	*М.3*	*М.4*
Full	**0.0040**	**0.0040**	**0.0041**	**0.0042**	**0.0040**
Full-non-rec	**0.0039**	**0.0040**	**0.0037**	**0.0039**	**0.0040**
Seq	0.0094	0.0099	0.0093	0.0096	0.0127
Non-rec	0.0100	0.0097	0.0096	0.0097	0.0097
Rec	0.0096	0.0096	0.0094	0.0099	0.0092
Min	0.0097	0.0097	0.0103	0.0095	0.0096
Max	0.0095	0.0110	0.0096	0.0094	0.0094
First	0.0099	0.0095	0.0096	0.0092	0.0098

Таблица 1 — Результат для `doubleAppendo` с конкатенацией трёх списков длины 120, секунды

	Б.С.	*М.1*	*М.2*	*М.3*	*М.4*
Full	0.614	0.634	**0.249**	**0.254**	**0.252**
Full-non-rec	0.611	0.607	0.594	**0.252**	**0.246**
Seq	**0.279**	**0.281**	0.280	0.279	0.280
Non-rec	**0.276**	**0.281**	0.277	0.275	0.281
Rec	0.857	0.867	0.560	0.562	0.560
Min	**0.280**	0.283	0.288	0.282	0.280
Max	**0.286**	0.282	0.279	0.281	0.277
First	0.861	0.868	0.564	0.948	0.950

Таблица 2 — Запуск для `maxLengtho` на списке `[1..200]`, секунды

	Б.С.	*М.1*	*М.2*	*М.3*	*М.4*
Full	0.260	0.259	0.263	0.258	**0.251**
Full-non-rec	0.255	**0.250**	0.259	0.258	**0.252**
Seq	**0.254**	**0.250**	**0.252**	**0.251**	0.264
Non-rec	0.256	0.262	0.260	0.255	0.263
Rec	0.261	0.259	0.263	0.266	0.268
Min	0.260	0.261	0.265	0.272	0.261
Max	0.268	0.257	0.273	**0.251**	0.257
First	0.255	0.261	0.263	0.256	0.260

Таблица 3 — Запуск для `sort`

В таблице 3 указаны результаты тестирования для программы `loginto`, которая использовалась для генерации всех выполнимых формул без свободных переменных.

Несмотря на то, что разные подходы на разных классах программ

	Б.С.	*М.1*	*М.2*	*М.3*	*М.4*
Full	-	-	0.1318	0.1049	-
Full-non-rec	**0.076**	0.0716	0.2097	0.1115	0.1258
Seq	0.168	0.1811	0.1491	0.0902	**0.0913**
Non-rec	0.078	0.0921	0.1356	0.0883	0.1144
Rec	0.081	0.0740	**0.0954**	**0.0637**	0.1263
Min	0.080	**0.0639**	0.1097	0.0921	0.1170
Max	0.164	0.1929	0.1438	0.0775	0.1108
First	0.181	0.1636	0.1757	0.0701	0.1894

Таблица 4 — Запуск для loginto **TODO: subst0**

	Б.С.	*М.1*	*М.2*	*М.3*	*М.4*
Full	-	-	0.078	0.068	-
Full-non-rec	0.056	0.045	0.125	0.084	0.082
Seq	0.109	0.110	0.086	0.063	0.074
Non-rec	**0.046**	**0.038**	0.081	0.072	**0.067**
Rec	0.055	0.050	0.074	**0.055**	0.079
Min	0.053	0.041	**0.066**	**0.055**	0.079
Max	0.100	0.117	0.108	0.057	0.074
First	0.118	0.103	0.091	0.068	0.101

Таблица 5 — Запуск для loginto **TODO: subst1**

	Б.С.	*М.1*	*М.2*	*М.3*	*М.4*
Full	-	-	0.078	0.062	-
Full-non-rec	0.137	0.040	0.093	0.042	0.069
Seq	0.086	0.082	0.066	0.049	**0.050**
Non-rec	0.043	**0.031**	0.063	0.044	0.055
Rec	**0.037**	0.034	**0.045**	0.040	0.051
Min	**0.037**	0.039	0.049	0.041	0.054
Max	0.068	0.070	0.067	**0.036**	0.062
First	0.104	0.100	0.110	0.095	0.137

Таблица 6 — Запуск для loginto **TODO: subst1**

показывают себя с лучшей стороны, в среднем модификация с обобщением вверх показывает себя лучше всего.

4.3 Сравнение суперкомпилятора с существующими решениями

В таблице 7

Параметр	*Оригинал*	*ECCE*	*CPD*	**Б.С**	**N.R.M.3**
doubleAppend	списки длины 120				
	0.0135	0.0051	0.0133	**0.0040**	0.0097
maxLength	список [1..200]				
	0.257	**0.230**	0.727	0.614	0.275
sort	случайный список длины 50				
	8.42	12.28	13.2	0.28	**0.25**
logint	размер подстановки				
0	> 300	0.17	2.7	-	**0.08**
1		0.09	1.7	-	**0.07**
2		0.08	0.9	-	**0.04**

Таблица 7 — Тестовые результаты, секунды

ЗАКЛЮЧЕНИЕ

В результате проделанной работы был разработан суперкомпилятор для miniKanren, реализованы его модификации, а также была произведена их апробация.

Реализованный суперкомпилятор показал улучшение производительности на подавляющем большинстве расмотренных программ относительно исходной программы, а также относительно реализаций конъюнктивной частичной дедукции, в иных случаях просадки производительности оказывались несущественными.

Исходный код проекта можно найти на сайте `https://github.com/RehMaar/uKanren-spec`, автор принимал участие под учётной записью RehMaar.

Результаты работы были представлены на конференции TEASE-LP'20.