СОДЕРЖАНИЕ

ВВЕДЕНИЕ

1 Описание предметной области

1.1 Логическое и реляционное программирование

Логическое программрование — 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* [41].

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 ресурсам [11].

Самые известные языки логического программирования — языки семества Prolog. Prolog применяется для доказательства теорем [13], проектирования баз знаний, создания экспертных систем [30] 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 [41].

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 [29] и задавало саму концепцию программы как отношения, когда логическое — появилось позже и предоставляло реализацию его идей [41]. 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.

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 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

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

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 вразумнительных 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[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.

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` расклады-

[2] `minikanren.org`

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

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

ваются на голову и хвост, а их хвосты состоят в отношении конкатенации с `Y`.

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

На рисунке 2 приведёт пример запроса, в котором мы хотим найти возможные значения переменных `Y` и `R`. Потенциально может быть бесконечное число ответов, к примеру, когда все аргументы в запросе — переменные, поэтому в системах miniKanren есть возможность запрашивать несколько первых ответов; в примере, это число 1. Ответы могут содержать в себе как конкретные замкнутые термы (к примеру, числа), так и свободные переменные, которые в примере обозначаются как $\_.n$. В примере одна и также свободная переменная $\_.0$ 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) (append^o [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[31], на данном этапе своего развития используется в основном в исследовательских целях:

– реляционные интерпретаторы на miniKanren для решения задач поиска [28], техника проргаммирования по примерам [5];
– для доказательство теорем [32] ;
– в области вычислительной лингвистики [39].

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[27], однако порождаемые им отношения — функциональные, а запуск их в обратном направлении, как указывалось выше, крайне непроизводителен[28].

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 [16].

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 [16].

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 [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 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 [23].

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 [26].

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 [25].

В работе [28] 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 [38].

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 [35]. 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*) [37].

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*) [34].

Техника суперкомпиляции в основном применяется для функциональных [2, 37] 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

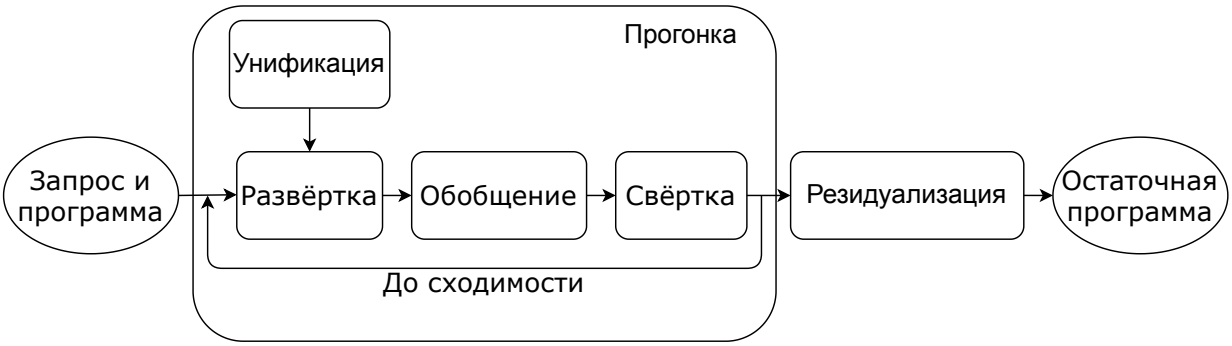

Рисунок 3 — Общая схема суперкомпилятора.

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[38].

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-

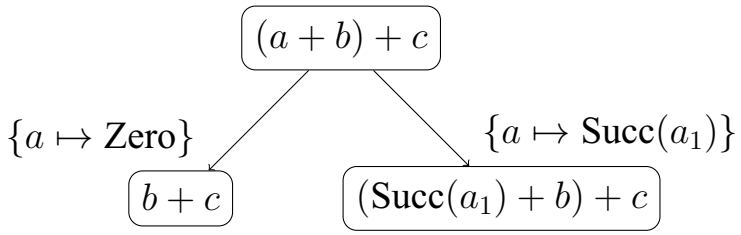

Рисунок 4 — Пример части графа процессов.

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 множно восстановить исходное дерево, используется *свёртка.*

Свёртка (англ. *folding*) — это процесс преобразования дерева процессов в граф, при котором из вершины v_c добавляется ребро в родительскую вершину v_p, если выражение в конфигурации в v_c и в v_p равны с точностью до переименования. Пример ситуации для свёртки изображён на рисунке 5, на котором свёрточное ребро изображено пунктиром.

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*[36]. Обобщение — это процесс замены одной конфигурации на другую, более абстрактную, описывающую больше состояний программы. Для обнаружения “похожей”

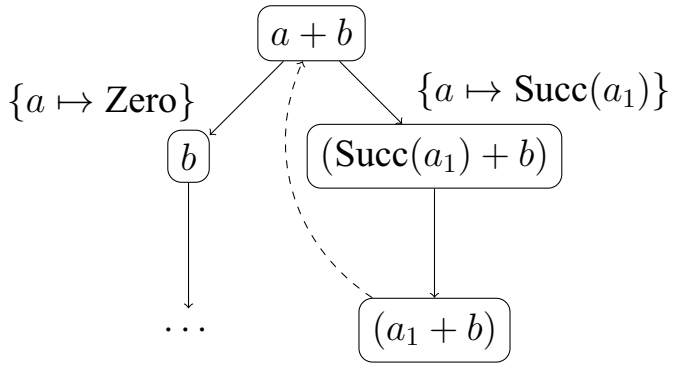

Рисунок 5 — Пример свёртки.

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

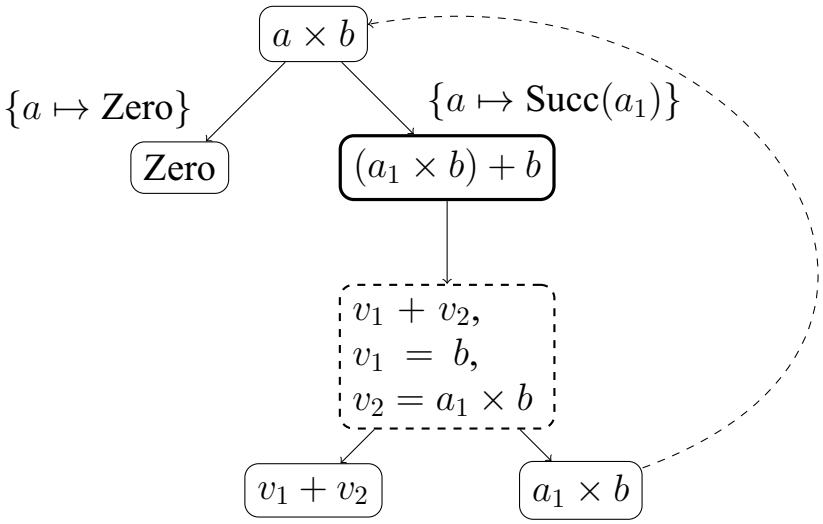

Рисунок 6 — Пример обобщения.

Построение программы по графу конфигураций называется *резидуализацией*, а построенная программа — *остаточной* (англ. *residual*). Ал-

горитм выявления остаточной программы основан на обходе дерева, но в остальном полностью зависит от языка.

2.2 Язык μKanren

В данной работе для специализации был выбран μKanren — минималистичный диалект языка miniKanren[12]. μKanren содержит только чистые операторы, что заначительно 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 [33].

– 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 [28].

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;

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

– 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': $\theta' = \sigma \circ \theta$[22]. 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 [37].

В-третьих, в качестве свистка используется отношение *гомеоморфного вложения*[36]. 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 [37]. 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[24]. 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 $e'_g \preccurlyeq e_g$[37]. 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.

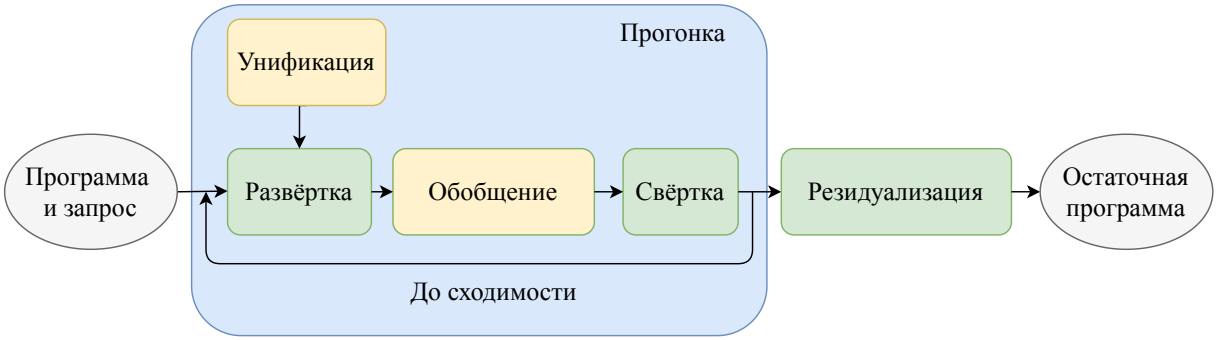

Рисунок 9 — Уточнённая схема суперкомпилятора.

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 [40]. Использование *IORef*[4], хотя и предоставляет изменяемность, 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 [20], 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 пере-

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

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

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

- `Failure` обозначает 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.

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[33], 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 по-

лучены новые конфигурации $f'(v_1) \wedge g(v_1) \wedge g''(v_2)$ и $f''(v_1) \wedge g(v_1) \wedge g''(v_2)$.

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

Таким образом, какой бы способ развёртывания определений 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.

3.2 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 части встречается, что в разных поддеревьях дерева процессов стречаются одинаковые конфигурации, поддеревья которых оказываются полностью идентичными. В таком случае, кажется очевидной оптмизация, при которой мы запоминаем вычисленные поддеревья и в случае, когда мы встречаем схожу конфигурацию, не вычисляем поддерево заново, добавляя ссылку на него. **TODO: Вопрос на засыпку меня же: может ли быть такое, что конфигурации являются вариантами друг друга, но накопленные подстановки приведут к по-**

явлению 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] — оптмизации, при которой множество проходов по одной структуре данных заменяется на один проход.

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

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

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

Рисунок 12 — Интерфейс для различных стратегий развёртывания.

Предоставляемые интерфейсом функции используются в алгоритме суперкомпиляции следующим образом:

– `initialize` оборачивает конфигурацию в структуру, в которой мо-

жет 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 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 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 поддерево родителя уничтожается.

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

```
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 *зипперы*[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 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.**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 15 представлено дерево процессов, при котором происходит обобщение 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

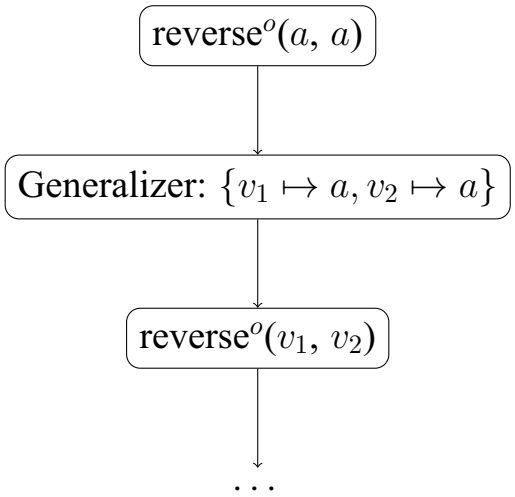

Рисунок 15 — Демонстрация потери информации при обобщении вверх.

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.

Расширение языка μKanren с помощью операции неэквивалентности

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

К примеру, на рисунке 16 изображена операция поиска значения по ключу в списке пар ключ-значения lookup^o.

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 унифика-

```
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.

ция `(K',V)::L'` $\equiv$ `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.

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

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

Исправленная версия отношения lookup^o представлена на рисунке 18.

```
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.

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

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

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

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

– `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 μKanren для тестирования использовался OCanren[5][21], встроенный в OCaml[21]. Для некоторых тестов для использовался faster-miniKanren[6], 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 μKanren библиотеки специализации.
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 сравивался с реализацией конъюнктивной частичной дедукции для μKanren[7], а также с реализацией конъюнктивной частичной дедукции для Prolog — системой ECCE[8]. Для последнего требовалось оттранслировать программу на μKanren 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 sort^o(list, result). Запросы:

[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`

  - оптимизация сортировки: sort^o(xs, ys);
  - генерация отсортированных последовательностей: sort^o(xs, xs).

- Отношение, проверяющее принадлежность пути графу isPath^o(path, graph, result). Специализация isPath^o(path, graph, true). Запросы:
  - генерация n произвольных путей в случайном графе;
  - поиск пути заданного размера в случайном графе: isPath^o_s(p, g)$\wedge$length^o(p, N).

- Интерпретатор формул логики высказываний logint^o(formula, subst, result). Запросы:
  - поиск n решений заданной формулы;
  - генерация n формул с подстановке размера n.

- Интерпретатор лямбда-исчисления lam^o(expr, result). Запросы:
  - генерация n выражений в нормальной форме lam^o(expr, expr);
  - генерация n выражений, редуцирующихся к заданному выражению lam^o(expr, E).

- Проверка типов в просто типизировнном лямбда-исчислении infer(type, expr)^o.
  - Поиск n обителей заданного типа.
  - Генерация выражений, соответствующих заданной спецификации типа и вырежения. Специализируется выражение: infer^o(type, expr) $\wedge$ type $\equiv$ TYPE_SPEC $\wedge$ expr $\equiv$ EXPR_SPEC.

- Интерпретатор простого подмножества Scheme.
  - **TODO: Интересный тест!**

ЗАКЛЮЧЕНИЕ

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

[1] Claire E. Alvis и др. “cKanren miniKanren with Constraints”. в: *In Proceedings of the 2011 Workshop on Scheme and Functional Programming (Scheme ’11), Portland, OR*. 2011.

[2] Maximilian Bolingbroke и Simon Peyton Jones. “Supercompilation by Evaluation”. в: *SIGPLAN Not.* 45.11 (сент. 2010), с. 135—146. ISSN: 0362-1340. DOI: 10.1145/2088456.1863540. URL: https://doi.org/10.1145/2088456.1863540.

[3] Ivan Bratko. *Prolog programming for artificial intelligence (4th ed.)* Harlow, England ; New York: Addison Wesley.

[4] William Byrd. “Relational Programming in miniKanren: Techniques, Applications, and Implementations”. в: (сент. 2009), с. 297.

[5] William E. Byrd и др. “A Unified Approach to Solving Seven Programming Problems (Functional Pearl)”. в: *Proc. ACM Program. Lang.* 1.ICFP (авг. 2017). DOI: 10.1145/3110252. URL: https://doi.org/10.1145/3110252.

[6] Danny De Schreye и др. “Conjunctive partial deduction: Foundations, control, algorithms, and experiments”. в: *The Journal of Logic Programming* 41.2-3 (1999), с. 231—277.

[7] Stephan Diehl. *A Prolog Positive Supercompiler*. 1997.

[8] Daniel P. Friedman, William E. Byrd и Oleg Kiselyov. *The Reasoned Schemer*. The MIT Press, 2005. ISBN: 0262562146.

[9] Yoshihiko Futamura. “Partial Evaluation of Computation Process - An Approach to a Compiler-Compiler”. в: *Systems, Computers, Controls* 2 (1999), с. 45—50.

[10] Robert Glück и Morten Heine Sørensen. “Partial Deduction and Driving are Equivalent”. в: *Lecture Notes in Computer Science*. Springer, Berlin, Heidelberg, 1994.

[11] M. Hanus, B. Peemöller и F. Reck. “Search Strategies for Functional Logic Programming”. в: *Proc. of the 5th Working Conference on Programming Languages (ATPS’12)*. Springer LNI 199, 2012, с. 61—74.

[12] Jason Hemann и Daniel P. Friedman. “μKanren: A Minimal Functional Core for Relational Programming”. в: *Proceedings of the 2013 Annual Workshop on Scheme and Functional Programming*. 2013.

[13] Jieh Hsiang и Mandayam Srivas. “Automatic inductive theorem proving using prolog”. в: *Theoretical Computer Science* 54.1 (1987), с. 3—28. ISSN: 0304-3975. DOI: https://doi.org/10.1016/0304-3975(87)90016-8. URL: http://www.sciencedirect.com/science/article/pii/0304397587900168.

[14] Zhenjiang Hu и др. “Tupling Calculation Eliminates Multiple Data Traversals”. в: *ACM SIGPLAN Notices* 32 (дек. 1997). DOI: 10.1145/258948.258964.

[15] Gérard Huet. “The Zipper”. в: *J. Funct. Program.* 7.5 (сент. 1997), с. 549—554. ISSN: 0956-7968. DOI: 10.1017/S0956796897002864. URL: https://doi.org/10.1017/S0956796897002864.

[16] Neil D Jones, Carsten K Gomard и Peter Sestoft. *Partial evaluation and automatic program generation*. Peter Sestoft, 1993.

[17] Jesper Jørgensen и Michael Leuschel. “Efficiently generating efficient generating extensions in Prolog”. в: *Partial Evaluation, International Seminar, LNCS 1110, pages 238–262, Schloß Dagstuhl*. Springer-Verlag, 1996, с. 238—262.

[18] Oleg Kiselyov и др. “Backtracking, Interleaving, and Terminating Monad Transformers: (Functional Pearl)”. в: *SIGPLAN Not.* 40.9 (сент. 2005), с. 192—203. ISSN: 0362-1340. DOI: 10.1145/1090189.1086390. URL: https://doi.org/10.1145/1090189.1086390.

[19] Andrei V. Klimov. *An Approach to Supercompilation for Object-oriented Languages: the Java Supercompiler Case Study*. 2008.

[20] Ilya Klyuchnikov и Sergei Romanenko. *SPSC: a Simple Supercompiler in Scala*. 2009.

[21] Dmitrii Kosarev и Dmitri Boulytchev. “Typed Embedding of a Relational Language in OCaml”. в: *Electronic Proceedings in Theoretical Computer Science* 285 (дек. 2018), с. 1—22. DOI: 10.4204/EPTCS.285.1.

[22] Jean-Louis Lassez, Michael Maher и Kim Marriott. “Unification Revisited”. в: т. 306. янв. 2006, с. 67—113. DOI: 10.1007/3-540-19129-1_4.

[23] Michael Leuschel. *Advanced Techniques for Logic Program Specialisation*. тех. отч. 1997.

[24] Michael Leuschel. “Homeomorphic embedding for online termination of symbolic methods”. в: *In The essence of computation, volume 2566 of LNCS*. Springer, 2002, с. 379—403.

[25] Michael Leuschel. *The ecce Partial Deduction System*. тех. отч. In Proc. of the ILPS’97 Workshop on Tools и Environments for (Constraint) Logic Programming, U.P, 1997.

[26] Michael Leuschel и др. “Specialising Interpreters Using Offline Partial Deduction”. в: т. 3049. янв. 2004, с. 340—375. DOI: 10.1007/978-3-540-25951-0_11.

[27] Boulytchev D. Lozov P. Vyatkin A. “Typed Relational Conversion”. в: *Wang M., Owens S. (eds) Trends in Functional Programming. TFP 2017. Lecture Notes in Computer Science, vol 10788. Springer, Cham* (2018).

[28] Petr Lozov, Ekaterina Verbitskaia и Dmitry Boulytchev. “Relational Interpreters for Search Problems”. в: *Relational Programming Workshop*. 2019, с. 43.

[29] Bruce J. MacLennan. *Introduction to relational programming*. тех. отч. 1981.

[30] Dennis Merritt. *Building Expert Systems in Prolog*. Spring-Verlag, 1989.

[31] Matthew Might. “The Algorithm for Precision Medicine (Invited Talk)”. в: *Proceedings Companion of the 2019 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity*. SPLASH Companion 2019. Athens, Greece: Association for Computing Machinery, 2019, с. 2. ISBN: 9781450369923. DOI: 10.1145/3359061.3365208. URL: https://doi.org/10.1145/3359061.3365208.

[32] Joseph Near, William Byrd и Daniel Friedman. “aleanTAP: A Declarative Theorem Prover for First-Order Classical Logic”. в: т. 5366. дек. 2008, с. 238–252. DOI: 10.1007/978-3-540-89982-2_26.

[33] Dmitry Rozplokhas и Dmitry Boulytchev. “Certified Semantics for miniKanren”. в: *miniKanren and Relational Programming Workshop*. 2019.

[34] Jens Peter Secher и Morten Heine Sørensen. “On perfect supercompilation”. в: *Journal of Functional Programming* 6 (1996), с. 465–479.

[35] Morten Heine Sørensen. *Turchin’s Supercompiler Revisited - An operational theory of positive information propagation*. 1996.

[36] Morten Heine Sørensen и Robert Glück. “An Algorithm of Generalization in Positive Supercompilation”. в: *ILPS*. 1995.

[37] Morten Heine Sørensen и Robert Glück. “Introduction to Supercompilation”. в: *Partial Evaluation - Practice and Theory, DIKU 1998 International Summer School*. Berlin, Heidelberg: Springer-Verlag, 1998, с. 246–270. ISBN: 3540667105.

[38] Valentin F. Turchin. “The Concept of a Supercompiler”. в: *ACM Transactions on Programming Languages and Systems* 8 (1986), с. 292–325.

[39] Zoltan Varju, Richard Littauer и Peteris Erins. “Using Clojure in Linguistic Computing”. в: февр. 2012.

[40] Илья Ключников. “Суперкомпиляция: идеи и методы”. в: *Практика функционального программирования, №7* (2011).

[41] Дж. Малпас. *Реляционный язык Пролог и его применение*. М.: Наука, 1990.