УНИВЕРСИТЕТ ИТМО
Факультет информационных технологий и программирования

Кафедра компьютерных технологий

Артемьева Ирина Александровна

Разработка транслятора из реляционного языка программирования в функциональный

Магистерская диссертация

Допущена к защите.
Зав. кафедрой:
д. ф.-м. н., профессор Омельченко А. В.

Научный руководитель:
Вербицкая Е. А.

Рецензент:
Березун Д. А.

Санкт-Петербург
2020

Оглавление

Введение

Реляционное программирование — парадигма, в которой любая программа описывает математическое отношение на её аргументах. Имея программу-отношение, можно выполнять запросы: указывая некоторые известные аргументы, получать значения остальных. Например, $add^o \subseteq Int \times Int \times Int$ описывает отношение, третий аргумент которого является суммой первых двух. Рассмотрим возможные направления вычисления этого отношения (здесь и далее искомый аргумент будем обозначать знаком "?"):

- $add^o\ x\ y\ ?$ с зафиксированными (входными) первым и вторым аргументом найдет их сумму;
- $add^o\ ?\ y\ z$ найдет такие числа, которые в сумме с y дадут z;
- $add^o\ ?\ ?\ z$ найдет такие пары чисел, что в сумме они равны z;
- $add^o\ ?\ ?\ ?$ перечислит все тройки из отношения.

Таким образом, мы можем говорить о выборе *направления* вычисления. Часто при написании программы подразумевается конкретное направление, называемое *прямым* (например, $add^o\ x\ y\ ?$), все остальные направления обычно называются *обратными*. Возможность выполнения в различных направлениях — основное преимущество реляционного программирования. Это своеобразный шаг к декларативности: достаточно написать одну программу для получения множества целевых функций.

Реляционному программированию родственно логическое, представленное такими языками, как Prolog и Mercury[1] [7]. Основным представителем парадигмы реляционного программирования является семейство интерпретируемых языков miniKanren[2]. Языки семейства miniKanren компактны и встраиваются в языки общего назначения, за счёт чего их

[1]Официальный сайт языка Mercury: `https://mercurylang.org/`, дата последнего посещения: 14.05.2020

[2]Официальный сайт языка miniKanren: `http://minikanren.org/`, дата последнего посещения: 14.05.2020

проще использовать в проектах. Для встраивания достаточно реализовать интерпретатор языка miniKanren: ядро языка, реализованное на Scheme занимает не более, чем 40 строк [2]. Помимо этого, miniKanren реализует полный поиск с особой стратегией, поэтому любая программа, написанная на нем, найдет все существующие ответы, в то время как Prolog может никогда не завершить поиск. В данной работе в качестве конкретного реляционного языка программирования используется miniKanren.

Возможность выполнения программ на miniKanren в различных направлениях позволяет решать задачи поиска посредством решения задачи распознавания [4]. Так, имея интерпретатор языка, можно решать задачу синтеза программ на этом языке по набору тестов [11]; имея функцию, проверяющую, что некоторая последовательность вершин в графе формирует путь с желаемыми свойствами, получать генератор таких путей и так далее. N-местную функцию-распознаватель, реализованную на некотором языке программирования, можно автоматически транслировать на miniKanren, получив $N + 1$-местное отношение, связывающее аргументы функции с булевым значением [4] (истина соответствует успешному распознаванию). Зафиксировав значение $N+1$-ого булевого аргумента, можно выполнять поиск. Ценность такого подхода в его простоте: решение задачи поиска всегда труднее, чем реализация распознавателя.

Однако, выполнение отношения в обратном направлении обычно крайне не эффективно. В [4] для решения этой проблемы используется специализация. В статье показано, что специализация приводит к существенному приросту скорости работы программы. Но, чтобы избавиться от всех накладных расходов, связанных с интерпретацией программы, необходим Джонс-оптимальный специализатор [3]. Он существует для Prolog [8], но не существует для miniKanren. В данной работе разрабатывается альтернативный подход улучшения производительности программы в заданном направлении — трансляция. Его суть — по отношению с фиксированным направлением генерируется функция на функциональном языке. В связи со сложностью задачи

транслятор, обеспечивающий прирост производительности, не разрабатывается сразу. Данная работа делает первый шаг к этому — разрабатывает алгоритм трансляции и проверяет, что транслированные программы корректны.

Кратко о следующих главах

- В главе 1 приводится цель работы и список необходимых для её достижения задач.
- В главе 2 даётся описание реляционного языка программирования miniKanren, формулируется задача трансляции, описывается анализ времени связывания как необходимая составная часть транслятора и даётся обзор существующих решений. В конце главы фомулируется цель данной работы, а также определяются задачи, решаемые в последующих главах.
- Глава 3 начинается с разбора особенностей трансляции. Далее приводится сам алгоритм трансляции. В конце доказывается сохранение семантики программ при трансляции.
- В главе 4 описывается разработка алгоритма аннотирования на основе анализа времени связывания. Вводится понятие нормализованной программы. Доказывается корректность предложенного алгоритма.
- Глава 5 посвящена тестированию работы алгоритма трансляции. В начале предлагается несколько классификаций программ на miniKanren и анализируются ограничения предложенного алгоритма. Затем описывается система его тестирования.

Список терминов и сокращений

Термины

interleaving парсер конъюнкт дизъюнкт

Сокращения

АСД – абстрактное синтаксическое дерево – в информатике конечное помеченное ориентированное дерево, в котором внутренние вершины сопоставлены (помечены) с операторами языка программирования, а листья – с соответствующими операндами.

ДНФ – дизъюнктивная нормальная форма – булевой логике нормальная форма, в которой булева формула имеет вид дизъюнкции конъюнкций литералов.

1. Цель и задачи работы

Целью данной работы является создание такого транслятора реляционного языка в функциональный, что транслированная функция обладает семантикой исходного отношения в выбранном направлении.

Для достижения этой цели решаются следующие задачи:

- Разработка алгоритма аннотирования, позволяющего транслятору определять направления и порядок вычислений конъюнктов.
- Разработка алгоритма трансляции реляционного языка в функциональный.
- Тестирование разработанного инструмента и анализ результатов.

2. Обзор предметной области

2.1. Язык программирования miniKanren

Семейство языков miniKanren дало рождение парадигме реляционного программирования. Это минималистичные языки, встраиваемые в языки программирования общего назначения. Как и Prolog они обладают свойством *полимодальности* — возможностью вычисляться в различных направлениях. Помимо простоты использования при разработке конечных приложений, miniKanren реализует полный поиск: все существующие решения будут найдены, пусть и за длительное время. Классический представитель родственной парадигмы логического программирования Prolog этим свойством не обладает: исполнение программы может не завершиться, даже если не все решения были вычислены. Незавершаемость программ на Prolog — свойство стратегии поиска решения. Для устранения потенциальной нетерминируемости используются нереляционные конструкции, такие как cut. Эта особенность существенно усложняет и часто делает невозможным исполнение в обратном направлении. Язык miniKanren же является чистым[3]: все языковые конструкции обратимы.

Программа на miniKanren состоит из набора определений отношений и цели. Определение имеет имя, список аргументов и тело. Тело отношения является *целью*, которая может содержать *унификацию термов* и *вызовы отношений*, скомбинированные при помощи *дизъюнкций* и *конъюнкций*. *Терм* представляет собой или *переменную*, или *конструктор* с именем и списком подтермов. Свободные переменные вводятся в область видимости при помощи конструкции $\underline{fresh}$. Абстракт-

[3]Технически это неправда: в miniKanren есть дополнительные конструкции, которые не являются реляционными. Однако, Ядро (или microKanren), при этом, чистое. В данной работе рассматривается только чистое подмножество.

ный синтаксис языка приведен ниже:

$$
\begin{aligned}
Goal &: Goal \vee Goal && (disjunction)\\
&\mid Goal \wedge Goal && (conjunction)\\
&\mid Term \equiv Term && (unification)\\
&\mid \underline{call}\ Name\ [Term]\\
&\mid \underline{fresh}\ [Var]\ Goal\\
Term &: Var\\
&\mid \underline{cons}\ Name\ [Term]
\end{aligned}
$$

Пример программы на языке miniKanren, связывающей три списка, где третий является конкатенацией первых двух, приведен на рисунке 2. Для краткости [] заменяет пустой список ($\underline{cons}\ Nil\ []$); $h : t$ обозначает список с головой h и хвостом t ($\underline{cons}\ Cons\ [h, t]$), а $[x_0, x_1, \ldots, x_n]$ — список с элементами $x_0, x_1, \ldots, x_n$. Вызов отношения $\underline{call}\ relation\ [t_0, \ldots t_k]$ записывается как $relation\ t_0 \ldots\ t_k$.

```
1  appendᵒ x y z =
2    (x ≡ [] ∧ y ≡ z) ∨
3    (fresh [h, t, r] (
4        x ≡ h : t ∧
5        z ≡ h : r ∧
6        appendᵒ t y r
7    ))
```

Рис. 1: Пример программы на miniKanren

Исполнение этого отношения в прямом направлении на двух заданных списках $append^o\ [1, 2]\ [3]$? вернёт их конкатенацию $[1, 2, 3]$. Если исполнить его в обратном направлении, оставив первые два аргумента неизвестными, мы получим все возможные разбиения данного списка на два: результатом $append^o\ ?\ ?\ [1, 2, 3]$ является множество пар $\{([], [1, 2, 3]), ([1], [2, 3]), ([1, 2], [3]), ([1, 2, 3], [])\}$. Стоит заметить, что miniKanren не всегда перечисляет все результаты, а иногда использует метапеременные.

2.2. Трансляция в функциональный язык

Суть трансляции: *По отношению с фиксированным направлением генерируется функция на функциональном языке программирования.*

Существуют трансляторы логических программ, но все они обладают спецификой Prolog или стараются сохранить полимодальность при трансляции. Так, в [6] обсуждается проблема трансляции *cut*-операции. [1] рассматривают способы интеграции логического и функционального программирования. Демонстрируется невозможность трансляции унификации в сопоставление с образцом по причине возможности унификации вычисляться в различных направлениях. В случае транслятора, предлагаемого в данной работе, направление быть должно и вышеупомянутой проблемы нет. Ту же проблему имеет [5] — попытка сохранить полимодальность при трансляции.

2.3. Анализ времени связывания

Особенностью реляционного программирования является отсутствие строгого порядка исполнения программы: он может отличаться для разных направлений. Это затрудняет трансляцию в функциональные языки программирования. Для успешной трансляции необходимо определить направления унификаций и вызовов отношений, а также их порядок исполнения с учётом направления трансляции. Для решения такой задачи используется *анализ времени связывания* (binding time analysis).

Анализ времени связывания разделяет программные конструкции на домены согласно моментам, когда конкретная конструкция получила связывание. В зависимости от цели применения могут выбираться разные домены времен связывания.

В данной работе цель — указать порядок, в котором имена связываются со значениями.

Анализ времени связывания часто используется при offline-специализации программ [3]. В этом случае он используется для определения того, какие данные известны статически и должны быть учтены при специали-

зации, а какие неизвестны. Также часто определяется, какие функции вообще следует специализировать и каким образом.

Анализ времени связывания существует для функционально-логического языка Mercury [12] и логического языка Prolog [9] — представителей родственных реляционному программированию парадигм. Однако, в языке Mercury анализ времени связывания [12] используется для эффективной компиляции. При этом используются только аннотации in и out — статические и динамические переменные. Этого недостаточно, чтобы определить порядок вычислений при трансляции в функциональный язык. Определение порядка вычислений в Mercury осуществляется во время более трудоемкого анализа модов (mode analysis), не существующего для miniKanren. При этом непосредственное использование этого подхода для miniKanren невозможно, так как не все языки семейства типизируемы, а анализ времени связывания Mercury осуществляется с учётом графа типов, построенного по программе.

Система LOGEN реализует анализ времени связывания для чистого подмножества Prolog [9]. Основное предназначение анализа в этой работе — улучшение качества специализации, упорядочивания вызовов не производится.

Работа [10] описывает анализ времени связывания для лямбда-исчисления с функциями высшего порядка. Его цель также в том, чтобы определить порядок связывания переменных, поэтому авторы используют отрезок натурального ряда $\{0, 1, \ldots, N\}$. Эта идея была использована в данной работе.

3. Разработка транслятора

Этот раздел посвящен разработке алгоритма трансляции miniKanren в абстрактный функциональный язык программирования. В первой части рассказывается об особенностях miniKanren и способах транслировать программу с этими особенностями. Вторая часть посвящена построению абстрактного синтаксическое дерева функционального языка с учётом особенностей трансляции miniKanren. Общее описание алгоритма трансляции находится в третьей части. Четвёртая часть посвящена доказательству сохранения семантики программы на miniKanren в наборе функций, полученных после работы алгоритма трансляции.

В нескольких частях примеры отсылаются к одному и тому же отношению $append^o$ (см. рисунок 2). Оно связывает три списка, первые два из которых являются конкатенацией третьего.

```
1  append^o x y z =
2    (x ≡ [] ∧ y ≡ z) ∨
3    (fresh [h, t, r] (
4        x ≡ h : t ∧
5        z ≡ h : r ∧
6        append^o t y r
7    ))
```

Рис. 2: Отношение $append^o$

3.1. Особенности miniKanren

miniKanren является реляционным языком программирования, а то время как трансляция осуществляется в функциональный. Данный раздел рассматривает особенности miniKanren и способы их поддержания в функциональном языке. Так как для тестирования абстрактный синтаксис функционального языка транслируется в конкретный синтаксис Haskell, описания решения проблем будут в терминах конструкций Haskell. Однако, это не умаляет общности получаемого транслятора. Используемые конструкции имеют аналоги в любом функциональном языке программирования, и можно написать транслятор аб-

страктного синтаксиса в любой конкретный.

Все рассматриваемые в данном разделе примеры являются результатом трансляции отношения $append^o$ в различных направлениях. Само отношение $append^o$ на miniKanren представлено на рисунке 2.

```
 8  append^o x^1 y^1 z^0 =
 9    (x^1 ≡ [] ∧
10     y^1 ≡ z^0) ∨
11    (fresh [h, t, r] (
12        x^3 ≡ h^1 : t^2 ∧
13        z^0 ≡ h^1 : r^1 ∧
14        append^o t^2 y^2 r^1
15    ))
```

Рис. 3: Проаннотированное в обратном направлении отношение $append^o$

3.1.1. Несколько выходных переменных

Не всегда результатом выполнения отношения является единственный ответ. Например, при выполнении отношения $append^o$ в обратном направлении, miniKanren вычислит *все* возможные *пары* списков, дающие при конкатенации z.

В общем случае отношению $R \subseteq X_0 \times \cdots \times X_n$, с известными аргументами $X_{i_0}, \ldots X_{i_k}$, и неизвестными $X_{j_0}, \ldots X_{j_l}$, соответствует функция, возвращающая список результатов $F : X_{i_0} \to \cdots \to X_{i_k} \to [X_{j_0} \times \cdots \times X_{j_l}]$.

В качестве решения проблемы предлагается использовать кортежи. Пример трансляции $append^o$ в обратном направлении приведён на рисунке 4. OOI рядом с названием функции обозначает направление трансляции отношения. Так, O — output и I — input. Пример получения кортежа в качестве результата находится в строке 7.

3.1.2. Пересечение результатов дизъюнктов

Дизъюнкты в программе на miniKanren независимы, то есть все ответы из каждого дизъюнкта объединяются для получения результата выполнения отношения. Чтобы поддержать данную особенность,

```
1   appendoOOI x0 = appendoOOI0 x0 ++ appendoOOI1 x0
2   appendoOOI0 s4@s0 = do
3     let s3 = []
4     return $ (s3, s0)
5   appendoOOI0 _ = []
6   appendoOOI1 s4@(s5 : s7) = do
7     (s6, s0) <- appendoOOI s7
8     let s3 = (s5 : s6)
9     return $ (s3, s0)
10  appendoOOI1 _ = []
```

Рис. 4: Результат трансляции отношения $append^o$ в обратном направлении

будем транслировать каждый дизъюнкт во вспомогательную функцию. Самому отношению будет соответствовать функция, конкатенирующая результаты этих вспомогательных функций. Стоит обратить внимание, что рекурсивно вызывается функция *appendoOOI*, построенная по всему отношению, а не какие-либо вспомогательные функции.

Примером такого поведения и соответствующей трансляции является $append^o$ в обратном направлении (см. рисунок 4. Здесь *appendoOOI* является основной функцией, объединяющей результаты вспомогательных функций *appendoOOI*0 и *appendoOOI*1. Если запустить *appendoOOI* на списке из трёх элементов [1, 2, 3], можно получить список всех пар списков, конкатенация которых даёт входящий список:
[([], [1, 2, 3]), ([1], [2, 3]), ([1, 2], [3]), ([1, 2, 3], [])]. Анализируя этот ответ легко понять, что первый кортеж получен из первого дизъюнкта (функции *appendoOOI*0), а все последующие — из второго (функции *appendoOOI*1).

3.1.3. Недетерминированность результатов

miniKanren способен выдавать несколько вариантов одной переменной в качестве результата, создавая недетерминированность. Монада списка — способ выразить недетерминированность в функциональном языке. Используем её в трансляции. Кроме того, в Haskell для

неё есть удобная нотация, называемая *do-нотацией*[4].

Её применение так же можно видеть на примере трансляции $append^o$ в обратном направлении, представленном на рисунке 4. Связывание в строке 7 означает, что результат будет вычислен для каждого элемента списка, полученного из рекурсивного вызова функции. Унификации неизвестных переменных (например $x \equiv []$ и $x \equiv h \,\%\, t$) при трансляции преобразуются в *let*-связывания (строки 3 и 8).

3.1.4. Переменные, принимающие все возможные значения

При вычислении отношений в различных направлениях нередко встречается ситуация, когда $fresh$-переменные, унифицирующиеся только друг с другом. В этом случае они остаются свободными. В miniKanren такие переменные могут принимать все допустимые значения.

Данная проблема является проблемой и для анализа времени связывания, обсуждаемого в следующем разделе. Результатом её разрешения является появление в дизъюнкте специальных унификаций вида $x \equiv < gen :>$, где x — переменная, оставшаяся свободной, а $< gen :>$ — нотация, позволяющая транслятору понять необходимость генерации. Таким образом, всё, что остаётся сделать на этапе трансляции — правильно раскрыть нотацию и произвести генерацию.

Рассмотрим пример трансляции $append^o\ x\ ?\ ?$ (см. рисунок 5). Генерация здесь происходит в строке 24 и выглядит как вызов функции, возвращающий список. Функция *gen* являющейся функцией класса типов *Generator*. Её реализация лежит на плечах пользователя: в зависимости от типа переменной она может генерировать списки различных сущностей в различном порядке. Реализация *Generator* для списков и цифр, а так же сам класс типов *Generator* представлены на рисунке 6.

3.1.5. Порядок и направление при исполнении отношений

Самая главная особенность — порядок и направление исполнения отношений внутри целевого может отличается для разных направле-

[4]Описание do-нотации языка Haskell: `https://en.wikibooks.org/wiki/Haskell/do_notation`, дата последнего посещения: 14.05.2020

```
22          appendoIOO x0 = appendoIOOO x0 ++ appendoIOO1 x0
23          appendoIOOO s0@[] = do
24            s1 <- (gen)
25            let s2 = s1
26            return $ (s1, s2)
27          appendoIOOO _ = []
28          appendoIOO1 s0@(s3 : s4) = do
29            (s1, s5) <- appendoIOO s4
30            let s2 = (s3 : s5)
31            return $ (s1, s2)
32          appendoIOO1 _ = []
```

Рис. 5: Результат трансляции отношения $append^o$ x ? ?

```
11          class Generator a where
12            gen :: [a]
13
14          instance (Generator a) => Generator [a] where
15            gen = [] : do
16              xs <- gen
17              x <- gen
18              return (x : xs)
19
20          instance Generator Int where
21            gen = [0..9]
```

Рис. 6: Класс типов *Generator* и его реализации для списков и цифр

ний. Так, отношение, выполненное в заданном направлении, можно рассматривать как функцию из известных аргументов в неизвестные. Например, отношение $append^o$, выполненное в прямом направлении, соответствует функции конкатенации списков x и y.

Отношение $append^o$ состоит из двух дизъюнктов. Первый дизъюнкт означает, что если x является пустым списком, то y совпадает с z. Второй дизъюнкт означает, что x и z являются списками, начинающимися с одного и того же элемента, при этом хвостом z является результат конкатенации хвоста списка x со списком y. Унификация с участием неизвестной переменной z указывает на то, *как* вычислить её значение, в то время как унификация известной переменной x — *при каком условии.*

Автоматическая трансляция $append^o$ в прямом направлении создаст функцию, приведенную на рисунке 8. В двух уравнениях первая пере-

менная сопоставляется с образцом. В первом случае мы сразу возвращаем второй список как результат, в то время как во втором необходимо осуществить рекурсивный вызов построенной функции.

```
8         appendoII0 x0 x1 = appendoII00 x0 x1 ++
              appendoII01 x0 x1
9         appendoII00 s3@[] s0 = do

10          let s4 = s0
11          return $ (s4)
12        appendoII00 _ _ = []

13        appendoII01 s3@(s5 : s6) s0 = do

14          (s7) <- appendoII0 s6 s0
15          let s4 = (s5 : s7)
16          return $ (s4)
17        appendoII01 _ _ = []
```

Рис. 7: Результат трансляции отношения $append^o$ в прямом направлении

Пример трансляции $append^o$ в обратном направлении приведен на рисунке 4.

Нетрудно заметить, что порядок вычислений в функциях не совпадает с порядком конъюнктов в исходном отношении. Например, рекурсивный вызов отношения $append^o$ производится в последнем конъюнкте (см. рис. 2, строка 6), в то время как в функциях выполняется в первую очередь.

Данная проблема была указана во введении как мотивация к разработке алгоритма аннотирования переменных. Адаптация анализа времени связывания к miniKanren и есть решение данной проблемы (рассматривается в следующем разделе). Рассмотрим конкретный пример, как аннотации переменных помогают выбрать порядок и направления вычислений отношений.

Пусть есть проаннотированное в обратном направлении отношение $append^o$ (см. рисунок 22), которое необходимо транслировать. Результат трансляции есть на рисунке 4.

Рассмотрим первый дизъюнкт. В начале определим направления вычислений каждой из них. Направление первой из них (в строка 9):

$let\ x\ =\ []$, так как аннотация константы всегда меньше аннотации переменной. Направление второй — $let\ y\ =\ z$, потому что аннотация z является 0 (входная переменная), в то время как y — 1. Таким образом, направление выбирается в соответствии с тем, какой части унификации принадлежит большая аннотация: кто содержит наибольшую, тому будет происходить присваивание. Теперь определим порядок. Для этого достаточно отсортировать получившиеся на прошлом шаге определения от меньшего к большему по аннотации определяемой переменной. В примере аннотации x и y совпадают и равны 1, поэтому в данном случае нам не важен их порядок. При трансляции (см. рисунок 4) определение y станет частью сопоставления с образцом (о том, как это работает, будет рассказано в следующем разделе) в строке 2, а определение x — строкой 3.

Перейдём ко второму дизъюнкту. Определим направления. Унификация в строке 11 обратится в определение $let\ x\ =\ (h\ :\ t)$, потому что аннотация x больше обоих аннотаций переменных h и t. Строка 12 даст определение $let\ (h\ :\ r)\ =\ z$. Направление вызова функции в строке 13 также определим по максимальной аннотации аргументов вызова. В данном случае она равна 2 и встречается у двух переменных — t и y. Это означает, что вызов происходит в том же направлении, что и исходное отношение, и направление будет выглядеть так: $(t,\ y) \leftarrow append^{o}OOI\ r$. Сортировка определений позволит получить их следующий порядок: $(h\ :\ r)$, $(t,\ y)$, x. При трансляции они окажется, соответственно, в 6, 7 и 8 строках.

3.2. Построение абстрактного синтаксическое дерева функционального языка

В предыдущей части были разобраны особенности трансляции, связанные с miniKanren. В этом — особенности, проявившиеся в процессе разработки алгоритма трансляции и повлиявшие на структуру абстрактного синтаксическое дерева функционального языка.

3.2.1. Сопоставление с образцом для входных переменных

Сопоставление с образцом — хороший способ отфильтровать заведомо ложные вычисления, используя информацию о типе конструктора аргумента. В качестве примера рассмотрим трансляцию $append^o$ в прямом направлении приведена на рисунке 8). Дизъюнкты исходного отношения $append^o$ (см. рисунок 2) содержат унификации первого аргумента, являющегося входным: первый дизъюнкт — унификацию с пустым списком (строка 9), второй — с непустым (строка 11). При трансляции такие унификации превращаются в сопоставление с образцом (см. строки 2 и 6). Как результат — если первый аргумент функции является пустым списком, то успешно вычислится только $appendoIIO0$.

```
1   appendoII0 x0 x1 = appendoII00 x0 x1 ++
        appendoII01 x0 x1
2   appendoII00 s3@[] s0 = do

3     let s4 = s0
4     return $ (s4)
5   appendoII00 _ _ = []

6   appendoII01 s3@(s5 : s6) s0 = do

7     (s7) <- appendoII0 s6 s0
8     let s4 = (s5 : s7)
9     return $ (s4)
10  appendoII01 _ _ = []
```

Рис. 8: Результат трансляции отношения $append^o$ в прямом направлении

Стоит отметить, зачем нужны строки 5 и 10. Они представляют собой сопоставление с образцом, которое всегда завершится успехом, однако, они не влияют на результат вычисления, так как возвращают пустой список. $appendoIIO1$, в случае не успешного сопоставления с образцом в строке 6, попытается найти уравнение, в котором сопоставление с образцом пройдёт успешно. Если строки 10 не будет, то вычисление функции $appendoIIO1$ завершится ошибкой за отсутствием возможности обработать соответствующий вход.

В случае, если в отношении на miniKanren одной переменной-аргументу соответствовало несколько унификаций, для неё появляется возмож-

ность выбрать, какая из них станет сопоставлением с образцом. Алгоритм выбирает ту из унификаций, которая обеспечит наибольшую вложенность конструкторов.

3.2.2. Совпадение имён в сопоставлениях с образцом

Имена переменных, используемых в сопоставлении с образцом, могут совпадать для разных аргументов. Например, во втором дизъюнкте $append^o$ на рисунке 2 есть унификации переменных x и z. Если мы будем транслировать $append^o\ x\ ?\ z$, то получим перекрытие имён переменных. Переменная h, участвующая в обеих унификациях, окажется и в обоих сопоставлениях с образцом: для первого и третьего аргументов.

На рисунке 9 представлен результат трансляции $append^o$ в обсуждаемом направлении. Второму дизъюнкту в нем соответствует функция $appendoIOI1$, а переменной h — переменная $s3$. Чтобы избежать перекрытия имён, $s3$ была переименована в $p2$ в сопоставлении с образцом для второго аргумента. Переименовывание нарушило условие, созданное при трансляции в данном направлении: оба аргумента-списка должны иметь одинаковый первый элемент списка. Восстановление этого условия происходит за счёт применения охранного выражения $\mid s3 == s2$.

```
11  appendoIOI x0 x1 = appendoIOI0 x0 x1 ++ appendoIOI1 x0 x1
12  appendoIOI0 s0@[] s2@s1 = return $ (s1)
13  appendoIOI0 _ _ = []
14  appendoIOI1 s0@(s3 : s4) s2@(p2 : s5) | s3 == p2 = do
15    (s1) <- appendoIOI s4 s5
16    return $ (s1)
17  appendoIOI1 _ _ = []
```

Рис. 9: Результат трансляции отношения $append^o\ x\ ?\ z$

Здесь же стоит заметить ещё одну особенность: рядом каждым сопоставлением с образцом существует его псевдоним. Этот псевдоним — исходное имя входной переменной то замены на сопоставление с образцом. Его необходимо сохранить для случая, если внутри тела функции потребуется именно эта переменная, а не переменные из сопоставления с образцом.

3.2.3. Совпадение имён в определениях

Переменные в определениях так же могут совпадать, однако, для них нельзя использовать охранные выражения. Использовать ветвление.

На рисунке 10 представлено модифицированное отношение $apppend^o$ (см. рисунок 2). Оно связывает три списка таких, что первый является повтором первого элемента второго списка, а третий — конкатенацией первого и второго списков.

```
8   appendoAssign x y z =
9     (x ≡ [] ∧
10     y ≡ z) ∨
11    (fresh [h, t, r, p, ps, c, cs] (
12        x ≡ h : t ∧
13        z ≡ h : r ∧
14        z ≡ p : (p : ps) ∧
15        z ≡ c : (c : cs) ∧
16        appendoAssign t y r
17    ))
```

Рис. 10: Отношение $append^o Assign$

Рассмотрим результат его трансляции в обратном направлении, представленный на рисунке 11). Строка 28 содержит определение, полученного из унификации $z \equiv c : (c : cs)$. Переменная c здесь стала переменной $s6$ и получили определение $let\ (s6 : (s6 : s7)) = s2$. Как и в случае аналогичной проблемы с сопоставлением с образцом, переименуем повторившуюая переменную. После чего необходимо добавить проверку на равенство исходной и переменной-замены. Все такие проверки накапливаются и происходят в конце — перед возвратом значения. Так, в 28 строке показано, что в случае невыполнения условия необходимо вернуть пустой список. Если условие выполняется, то возвращается результат.

3.2.4. Трансляция конструкторов

Терм miniKanren может быть произвольным конструктором. В этом случае чтобы успешно выполнить полученную после трансляции функ-

```
18          appendoAssignOOI x0 = appendoAssignOOI0 x0 ++ appendoAssignOOI1 x0
19          appendoAssignOOI0 s2@s1 = do
20            let s0 = []
21            return $ (s0, s1)
22          appendoAssignOOI0 _ = []
23          appendoAssignOOI1 s2@(s8 : (p2 : s9)) | s8 == p2 = do
24            let (s3 : s5) = s2
25            let (s6 : (c4 : s7)) = s2
26            (s4, s1) <- appendoAssignOOI s5
27            let s0 = (s3 : s4)
28            if (s6 == c4) then return $ (s0, s1) else []
29          appendoAssignOOI1 _ = []
```

Рис. 11: Результат трансляции отношения $append^oAssign$ в обратном направлении

цию, необходимо знать, как вычислять данный конструктор. При трансляции будем считать, что пользователь сам позаботится о способе вычисления конструктора. Так как конструктор является функцией, достаточно её определить.

На рисунке 12 приведен один из таких пользовательских конструкторов, реализующих натуральные числа.

```
30          data Peano = O | S Peano
31
32          p2i :: Peano -> Int
33          p2i O     = 0
34          p2i (S x) = succ $ p2i x
35
36          i2p :: Int -> Peano
37          i2p 0 = O
38          i2p n | n < 0     = O
39                | otherwise = S (i2p $ pred n)
```

Рис. 12: Тип данных $Peano$ — определение конструкторов O и S

3.2.5. Абстрактный синтаксис функционального языка

Транслированная программа $FuncProgram$ представляет собой множество функций F.

Каждая функция F определяется именем и списком вспомогательных функций $Line$.

Каждая вспомогательная функция *Line* состоит из списка сопоставлений с образцом *Pat* (представляющего собой список аргументов), списка охранных выражений *Guard* для сопоставления с образцом, списка определений *Assign*, списка охранных выражений *Guard* для определений и значения выражения.

Сопоставление с образцом *Pat* состоит из опционального псевдонима и тела сопоставления с образцом, называемого в данном случае *Atom*.

Atom является аналогом *Term* из miniKanren с небольшим расширением. *Atom* может быть переменной или конструктором, однако, ещё он может быть кортежем (конструктор *Tuple*). Кореж — список переменных без конструктора. Используется, когда необходимо вернуть несколько переменных после вызова функции.

Охранное выражение *Guard* — список *Atom*, которые необходимо проверить друг с другом на равенство.

Определение *Assign* представляет собой *Atom* и *Expr* — значение выражения *Expr* будет сопоставлено *Atom*.

Выражение *Expr* может быть или тоже *Atom*, или вызовом функции на списке *Atom*.

```
30  data Atom = Var String
31            | Ctor String [Atom]
32            | Tuple [String]
33
34  data Expr = Term Atom
35            | Call String [Atom]
36
37  data Assign = Assign Atom Expr
38
39  newtype Guard = Guard [Atom]
40
41  data Pat = Pat (Maybe String) Atom
42
43  data Line = Line [Pat] [Guard] [Assign] [Guard
        ] Expr
44
45  data F = F String [Line]
46
47  newtype FuncProgram = FuncProgram [F]
```

Рис. 13: Абстрактный синтаксис функционального языка

3.3. Алгоритм трансляции

Данная часть приводит общий алгоритм трансляции программы в абстрактном синтаксисе miniKanren в программу в абстрактном синтаксисе функционального языка.

Первым шагом алгоритма трансляции является запуск алгоритма аннотирования, в процессе которого будут произведено приведение программы к нормальной форме. Таким образом, из произвольной программы на miniKanren будет получен стек всех вызовов этой программы, каждый вызов в котором — нормализованное проаннотированное в определённом направлении определение на miniKanren.

Далее для каждого дизъюнкта каждого определения со стека вызовов запускается сам алгоритм трансляции. В первом приближении он состоит из следующих шагов:

- Формирование списков входных и выходных переменных *in* и *out*;
- Разбиение конъюнктов на те, которые могут стать сопоставлениями с образцом и все остальные;
- Удаление перекрытий имён в сопоставлении с образцом путём формирования охранных выражений;
- Определение направлений конъюнктов;
- Определение порядка определений;
- Удаление перекрытий имён в определениях путём формирования условий для ветвления.

Многие из этих шагов были описаны в предыдущих частях об особоенностях miniKanren и построении абстрактного синтаксического дерева функционального языка. Подробнее рассмотрим только два шага — получение направлений конъюнктов и получение порядка определений.

Получение направлений конъюнктов происходит по-разному для унификаций и вызовов. Если конъюнкт — унификация, определим значе-

ние максимальной аннотации в каждой из её частей. Теперь возможны три случая:

- Если эти значения равны, то унификация становится условием ветвления;
- Если значение левой части больше, то унификация превращается в определение, где левая часть зависит от правой; в ответ сохраняем не только получившееся определение, но и максимальную аннотацию левой части — это необходимо для получения порядка вычисления определений;
- Оставшийся случай симметричен;

Если конъюнкт — вызов отношения, определим его выходные переменные. Для этого достаточно узнать максимальную аннотацию его аргументов — переменные обладающие таковой, стали известны после выполнения вызова и являются зависимыми. Также, как делали для унификаций, сохраним значение максимальной аннотации.

Получим порядок определений. Для этого отсортируем их по максимальной аннотации их зависимой части.

3.4. Доказательство сохранения семантики транслированной программы

Трансляция считается корректной, если семантика полученной после трансляции функции совпадает с семантикой исходного отношения в заданном направлении. В целом, для этого необходимо перебрать все программы на miniKanren, применить к ним алгоритм трансляции и проверить, изменилось ли семантика.

4. Анализ времени связывания

Данный раздел посвящен анализу времени связывания для miniKanren В первой части рассказано об адаптации идей анализа времени связывания для определения направления вычислений. Вторая часть вводит понятие программы в нормальной форме, необходимой для анализа времени связывания. В этой же части акцентирует внимание на отличиях нормализованной программы от ненормализованной. Алгоритм аннотирования для программ в нормальной форме описывается в третьей части. Несколько примеров аннотирования — в четвёртой части. Пятая часть предлагая способы нормализации программ с учётом необходимости последующего аннотирования. В шестой части представлена корректность предложенного алгоритма.

4.1. Анализ времени связывания для miniKanren

Цель анализа времени связывания — указать порядок, в котором имена связываются со значениями. Алгоритм принимает на вход программу на miniKanren и данные о том, какие переменные считаются входными. В результате работы алгоритма каждой переменной ставится в соответствие положительное число, обозначающее время связывания этой переменной. Мы будем называть процесс подбора чисел *аннотированием*, а сам алгоритм — алгоритмом анализа времени связывания или алгоритмом аннотирования.

Если о переменной ничего неизвестно, она аннотируется $Undef$; иначе указывается время связывания: целое положительное число. В начале работы алгоритма известными являются переменные, указанные как входные — они аннотируются числом 0. Если переменная унифицируется с константой (термом, не содержащим свободных переменных), то мы считаем её временем связывания 1. Если переменная унифицируется с термом, каждая свободная переменная которого аннотирована, мы аннотируем эту переменную числом $1 + n$, где n — максимальная аннотация свободных переменных терма. Таким образом мы распространяем информацию о времени связывания на непроаннотированные

переменные.

На аннотациях имеется порядок — естественный порядок на положительных числах, при этом $Undef$ считается меньше любой числовой аннотации. Ранее проаннотированная переменная может получить другую аннотацию, если появилась какая-то новая информация о её времени связывания. При этом аннотация никогда не заменяется на меньшую.

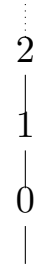

Рис. 14: Полурешетка на аннотациях

4.2. Понятие нормальной формы

Любое отношение miniKanren можно преобразовать в нормальную форму. *Нормальной формой* будем называть дизъюнкцию конъюнкций вызовов отношений или унификаций термов, в которой все свободные переменные введены в область видимости в самом начале; при этом отсутствуют унификации двух конструкторов. Соответствующий абстрактный синтаксис приведен на рисунке 15.

Рассмотрим отличия нормализованной программы от ненормализованной. Для каждого из них обсудим причины появления и способ получения из ненормализованной программы.

- Тело определения находится в дизъюнктивной нормальной форме;
- Все свободные переменные введены при помощи $fresh$ на самом верхнем уровне;
- Не существует унификаций термов-конструкторов;

$$
\begin{aligned}
Goal &: \underline{fresh}\ [Name]\ (\bigvee \bigwedge Goal') \\
Goal' &: \underline{call}\ Name\ [Var] \\
&\mid Var \equiv Term \\
Term &: Var \\
&\mid \underline{cons}\ Name\ [Term]
\end{aligned}
$$

Рис. 15: Абстрактный синтаксис нормализованной программы на miniKanren

- Не существует вызовов на термах-конструкторах.

Такие ограничения вводятся с целью упрощения процесса аннотирования и трансляции в целом:

- ДНФ тела позволяет уменьшить глубину вложенности программы;
- $fresh$-цель задаёт область видимости вычислений и позволяет использовать одинаковые имена переменных в различных областях видимости — её наличие только на самом верхнем уровне означает, что все переменные принадлежат одной области видимости;
- Отсутствие унификаций термов-конструкторов позволяет не производить очевидной унификации в процессе выполнения алгоритма;
- Отсутствие вызовов на термах-конструкторах позволяет избежать неопределённости в процессе аннотирования.

Если в программе на miniKanren отсутствуют вызовы на термах-конструкторах, то привести её к нормальной форме несложно:

- Приведение булевого выражения в дизъюнктивную нормальную форму — тривиальная задача;

- Если уникально переименовать все $fresh$-переменные отношения, то fresh-цель можно оставить только на самом верхнем уровне, избежав перекрытия имён;

- Унификацию термов-конструкторов, если совпадают их имена и количество аргументов, всегда можно заменить на унификацию переменной и терма;

- Способ аннотирования программ с вызовами на термах-конструкторах рассматривается в последующих частях.

4.3. Алгоритм аннотирования нормализованной программы

Алгоритм аннотирования получает на вход нормализованную программу на miniKanren (цель и список определений), а также список входных переменных. По окончанию его работы будет получен список проаннотированных определений, требуемых для вычисления цели. Мы будем называть этот список *стеком вызовов*, потому что в нем будут находиться вызываемые отношения.

Успешным результатом аннотирования назовём ситуацию, когда получившийся по окончании выполнения алгоритм стек вызовов удовлетворяет следующим условиям:

- Все отношения, требуемые для вычисления цели программы, присутствуют в стеке;

- Все переменные отношений, присутствующих в стеке вызовов, проаннотированы числом;

При инициализации алгоритма выполняются следующие действия:

- Все входные переменные аннотируются 0;

- Создается пустой стек вызовов;

Аннотация цели осуществляется итеративно, пока не будет достигнута неподвижная точка функции, описывающей шаг аннотирования. За один шаг аннотируется хотя бы одна унификация или один вызов отношения. Если в течение шага ни одна новая переменная не была проаннотирована, считается, что достигнута неподвижная точка. Для аннотации цели в дищъюнктивной нормальной форме необходимо проаннотировать все её дизъюнкты. Аннотации переменных в дизъюнкте должны согласовываться: одна и та же переменная в конъюнктах одного дизъюнкта должна иметь одну и ту же аннотацию. Конъюнкты аннотируются в заранее определенном порядке. Сначала мы аннотируем унификации, а затем вызовы отношений. Каждый раз при аннотации новой переменной необходимо установить ту же аннотацию всем другим вхождениям этой переменной в дизъюнкте.

При аннотировании унификаций возможны следующие случаи. Здесь и далее аннотация переменной указывается в верхнем индексе.

- Унификация имеет вид $x^{Undef} \equiv t[y_0^{i_0}, \ldots, y_k^{i_k}]$, то есть переменная, имеющая аннотацию $Undef$, унифицируется с термом t со свободными переменными $y_j^{i_j}$ с целочисленными аннотациями i_j. В таком случае переменной x необходимо присвоить аннотацию $n+1$, где $n = max\{i_0, \ldots i_k\}$;

- Переменная, аннотированная числом, унифицируется с термом: $x^n \equiv t[y_0^{i_0}, \ldots, y_k^{i_k}]$; некоторые свободные переменные терма проаннотированны $Undef$. Тогда всем переменным y_j^{Undef} присваивается аннотация $n+1$;

- Остальные случаи симметричны;

Помимо унификации конъюнкт может быть вызовом некоторого отношения. Если переменные всех аргументов проаннотированы $Undef$, для аннотирования не достаточно информации, поэтому следует перейти к аннотации следующего конъюнкта. Если хотя бы одна переменная-аргумент не $Undef$, произведём аннотацию вызова. Она состоит из двух

частей: аннотации аргументов самого вызова и, в случае необходимости, аннотации тела вызываемого отношения в соответствии с направлением вызова.

Аннотация тела вызываемого отношения состоит из следующих шагов:

- Получение направление вызова. Для этого аннотации аргументов "сбрасываются": $Undef$ остаются таковыми, а числовые — становятся 0. Для вызываемого отношения не важен момент времени в прошлом, когда его входные переменные стали известны — для него они все стали известны в момент времени 0;
- Аргументы вызова подставляются вместе со "сброшенными" аннотациями в тело вызываемого отношения;
- Имя, направление вызова и частично проаннотированное тело помещаются в стек вызовов;
- Происходит запуск алгоритма аннотирования;
- Обновляется стек вызовов: по имени и направлению помещается тело вызова после аннотирования;

Добавление в стек вызовов информации о ранее проаннотированных в конкретных направлениях отношениях позволяет избежать повторного аннотирования. В частности, помогает не получить бесконечный цикл при аннотировании рекурсивного вызова. Как это происходит: если частично определенное направление текущего вызова согласовано с ранее проаннотированным, анализировать его не нужно. Два направления назовем *согласованными*, если:

- Аннотации их аргументов попарно совпадают;
- Некоторые аннотации аргументов одного из направлений являются $Undef$, а оставшиеся, числовые, совпадают с соответствующими числами аннотаций аргументов другого направления;

Для иллюстрации понятия согласованных направлений рассмотрим следующие примеры:

- Пусть есть отношение r^o с частично определенными направлениями: $r^o\ x^0\ y^0\ z^{Undef}$ и $r^o\ x^1\ y^0\ z^{Undef}$; Они являются несогласованными, так как аннотации переменной x являются числовыми и не совпадают.

- Направление $r^o\ x^1\ y^0\ z^{Undef}$ согласовано с направлением $r^o\ x^1\ y^0\ z^1$, так как аннотация z в первом направлении является $Undef$ и, значит, может оказаться как входной, так и выходной.

Перейдём к аннотации аргументов самого вызова. Первым шагом нужно определить, есть ли необходимость аннотировать тело вызова. По имени и направлению проверяем наличие согласованного направления в стеке вызовов. Если такового не оказалось, запустим аннотацию тела вызываемого отношения, а иначе сразу перейдём к аннотации аргументов. Для этого необходимо заменить $Undef$-аннотации переменных на $n+1$, где n — максимальная аннотация переменных-аргументов вызова.

Существуют отношения, точная аннотация которых не возможна без полного перебора возникающих вариантов. Один из видов таких отношений — отношения, содержащие несколько вызовов в одном дизъюнкте Пример такого отношения приведен на рисунке 16. Пусть y — входная переменная. В этом случае порядок вычисления вызовов f^o и h^o не зависит друг от друга, но зависит от направления вычисления g^o. Оно не может вычисляться до вычисления f^o и h^o (неизвестны входные переменные), но может вычисляться между ними (в прямом или обратном порядке) или после (выполнять роль предиката).

Рассмотрим конкретный пример. Возьмём то же определение $revers^o$, что было рассмотрено на рисунке 23, и попробуем проаннотировать его в прямом направлении (см. рисунок 17).

Во втором дизъюнкте определим аннотации переменных h и t в строке 29 и распространим их на последующие вызовы. Зная, что только второй аргумент входной, попытаемся проаннотировать тело $append^o$

```
22  rel^o x y z =
23      f^o x y ∨
24      h^o z y ∨
25      g^o x z
```

Рис. 16: Пример программы на miniKanren с несколькими вызовами в одном дизъюнкте

(см. рисунок 18). Оно завершится неудачией: строках 36 и 36 остались переменные, проаннотированные $Undef$. Причина в том, что при таком направлении не существует возможности узнать значения (а, значит, и время связывания) переменных h, x и z (данная проблема обсуждается в следующем подразделе). Мы можем успешно завершить аннотирование $revers^o$, однако, при трансляции не сможем получить работающую программу, так как не успешно завершилось аннотирование $append^o$.

```
26  revers^o x^0 y^1 =
27      (x^0 ≡ [] ∧ y^1 ≡ []) ∨
28      (fresh [h, t, r] (
29          x^0 ≡ h^1 : t^1 ∧
30          append^o r^2 [h^1] y^2
31          revers^o t^1 r^2 ∧
32      ))
```

Рис. 17: Результат аннотирования отношения $revers^o$ в прямом направлении с порядком вызовов $append^o$-$revers^o$

```
33  append^o x^1 y^0 z^1 =
34      (x^1 ≡ [] ∧ y^1 ≡ z^0) ∨
35      (fresh [h, t, r] (
36          x^Undef ≡ h^Undef : t^1 ∧
37          z^Undef ≡ h^Undef : r^1 ∧
38          append^o t^1 y^0 r^1
39      ))
```

Рис. 18: Результат аннотирования отношения $append^o$? y ?

Посмотрим, как будет происходить аннотирование, если поменять местами вызовы $append^o$ и $revers^o$ 19. Строка 44 содержит рекурсивный вызов того же направления, что и исходное отношение — так становится

известна переменная r. Аннотирование будет успешно завершено аннотированием вызова $append^o$ в обратном направлении (см. рисунок 22) в строке 45.

```
40  revers^o x^0 y^1 =
41    (x^0 ≡ [] ∧ y^1 ≡ []) ∨
42    (fresh [h, t, r] (
43        x^0 ≡ h^1 : t^1 ∧
44        revers^o t^4 r^3 ∧
45        append^o r^3 [h^2] y^0
46    ))
```

Рис. 19: Результат аннотирования отношения $revers^o$ в прямом направлении с порядком вызовов $revers^o$-$append^o$

Для решения проблемы нескольких вызовов в одном дизъюнкте предложено решение с перестановками конъюнктов. Если в аннотируемом отношении в одном дизъюнкте найдено несколько вызовов, создаётся несколько версий этого дизъюнкта. Каждая версия отличается очередной перестановкой вызовов. Далее происходит запуск аннотирования дизъюнкта на каждой из версий до тех пор, пока либо аннотирование закончится успехом, либо будут просмотрены все возможные версии. В последнем случае считается, что аннотирование не успешно.

Причина, по которой невозможно проаннотировать $append^o$? y ? (см. рисунок 18) — $fresh$-переменные, которые зависят только друг от друга. В данном примере это переменные h, x и z. h встречается в двух унификациях, но проаннотировать её невозможно. В строке 36 она унифицируется с переменной x, а в строке 37 — с переменной z. x, и z являются выходными (fresh-переменными) и их значения остаются неизвестными, так как они не присутствуют в последнем конъюнкте (строка 38).

Таким образом, не существует возможности проаннотировать $fresh$-переменные, зависящие друг от друга, так как они никогда не станут известны — остаются свободными. В miniKanren такие переменные могут принимать все допустимые значения. Мы можем сымитировать данный подход, добавив генерацию переменных оставшихся свободными. Под *добавлением генерации* понимается добавление в дизъ-

юнкт нового конъюнкта — унификации целевой переменной со специальным термом-конструктором вида C "gen" [] (в конкретном синтаксисе $< gen :>$). В этом случае аннотация такой переменной будет являться аннотацией константы и равняться 1.

Рассмотрим полный алгоритм добавления генерации. Прежде всего, необходимо произвести аннотирование отношения без генерации по вышеописанному алгоритму, чтобы выяснить, какие переменные не получилось проаннотировать. Если полученный после аннотирования стек содержит частично проаннотированные определения, проанализируем каждое из них. Рассмотрим каждую унификацию. Если она содержит непроаннотированные переменные, то содержит их в обеих частях. Нет смысла генерировать переменные обеих частей: переменные одной части станут известны, если станут известны переменные другой. Поймём, какая из частей унификации является более частным случаем и будем генерировать переменные именно этой части.

Посмотрим, как по двум термам определить, является ли первый из них частным случаем второго. Алгоритм проинициализируем пустой подстановкой — списком пар переменная-терм. Если первый терм является частным случаем второго, то результатом алгоритма будет подстановка соответствующих подтермов второго терма вместо переменных первого терма. Возможным следующие варианты:

- Первый терм — переменная v, второй — терм u (может быть и переменной, и конструктором). По переменной v попробуем получить терм подстановки. Если это удалось сделать, сравним полученный терм с u — в случае совпадения вернём текущую подстановку, иначе искомой подстановки не существует и первый подтерм не является подтермом второго. Если v в подстановке не нашлось, добавим в неё пару $(v,\ u)$.

- Случая двух конструкторов нет для нормализованной программы нет.

- В оставшихся случаях искомой подстановки заведомо не существует.

После добавления генерации необходимо запустить алгоритм аннотирования ещё раз, чтобы проаннотировать сгенерированные переменные и распространить их аннотации на всё отношение. Шаг "генерация-аннотация" нужно повторять до достижения неподвижной точки: после генерации при повторном аннотировании может появиться возможность проаннотировать вызов, который до этого был на полностью неопределённых переменных. Для аннотирования тела этого вызова так же может потребоваться генерация. Аннотирование $append^o\ ?\ y\ ?$ с добавлением генерации приведено на рисунке 20.

```
33  append^o x^1 y^0 z^1 =
34    (x^1 ≡ [] ∧ y^1 ≡ z^0) ∨
35    (fresh [h, t, r] (
36         h^1 ≡ < gen :> ∧
37         x^2 ≡ h^1 : t^1 ∧
38         z^2 ≡ h^1 : r^1 ∧
39         append^o t^1 y^0 r^1
40    ))
```

Рис. 20: Результат аннотирования отношения $append^o\ ?\ y\ ?$ с добавлением генерации

Генерация позволила проаннотировать $append^o\ ?\ y\ ?$. В этом случае аннотирование $revers^o$ с последовательностью вызовов $append^o$-$revers^o$ (рисунок 17) становится успешным. Это даёт два способа трансляции $revers^o$ в прямом направлении. Так, для примера на рисунке 17 из вызовов будут сгенерированны две функции: предикат $revers^o$ и $append^o\ ?\ y\ ?$. Для отношения на рисунке 19) — только $append^o$ в обратном направлении.

Генерация переменных способна влиять на направления вычислений конъюнктов, поэтому её стоит применять только по необходимости. Чтобы нивелировать это влияние, будем генерировать переменные только в случае не успешного аннотирования. Для этого запустим алгоритм генерации после запуска алгоритма перебора перестановок конъюнктов. На примере $revers^o$: из двух вариантов предпочтительной последовательностью конъюнктов обладает вариант на рисунке 19 — $revers^o$ с последовательностью вызовов $revers^o$-$append^o$. Аннотиро-

вание второго дизъюнкта на рисунке 17 завершится неудачей, поэтому произойдёт перестановка вызовов и получим последовательность конъюнктов другого варианта. Это приведёт к успеху аннотирования и генерация не понадобится.

4.4. Примеры аннотирования

В этом разделе приведено несколько примеров аннотирования отношений. Числа над переменными обозначают аннотации.

4.4.1. Отношение $append^o$ в прямом направлении

$append^o$ — отношение связывающее три списка, первые два из которых являются конкатенацией третьего. Его аннотирование в прямом направлении представлено на рисунке 21.

В данном случае переменные x и y являются входными. При начале работы алгоритма, таких отношения и направления нет в стеке вызовов, поэтому добавим их и запустим рекурсивно аннотирование цели $append^o$. Так как x и y — входные переменные, их аннотации нам известны и равны 0.

Рассмотрим аннотирование первого дизъюнкта. x и y известны — остаётся определить z. Аннотация z равна 1, так как z унифицируется с y, аннотация которой — 0.

Во втором дизъюнкте аннотации h и t в строке 4 можно установить, так как известна аннотация x. Аннотация h распространяется на 5 строку, а аннотация t — на 6 строку. Рекурсивный вызов отношения в строке 6 согласован с имеющимся в стеке, поэтому можно проаннотировать переменную r. Распространяем аннотацию r в строке 5. На последнем шаге аннотируем z в строке 4.

4.4.2. Отношение $append^o$ в обратном направлении

Теперь рассмотрим аннотирование $append^o$ в обратном направлении. В этом случае мы считаем переменную z входной (см. рисунок 22). Пусть $append^o$ уже в стеке и z проаннотирована. В первом дизъюнкте

```
1   append^o x^0 y^0 z^1 =
2     (x^0 ≡ [] ∧ y^0 ≡ z^1) ∨
3     (fresh [h, t, r] (
4         x^0 ≡ h^1 : t^1 ∧
5         z^3 ≡ h^1 : r^2 ∧
6         append^o t^1 y^0 r^2
7     ))
```

Рис. 21: Результат аннотирования отношения $append^o$ в прямом направлении

x и y имеют аннотацию 1: y унифицируется со входной переменной z, а x — с константой. Во втором дизъюнкте на первом шаге становятся известны аннотации h и r (строка 12). Аннотация r распространяется на строку 13. Отношение с согласованным направлением есть в стеке, поэтому можно аннотировать t и y. Далее аннотация t распространяется на строку 11, и на последнем шаге аннотируется x.

```
8   append^o x^1 y^1 z^0 =
9     (x^1 ≡ [] ∧ y^1 ≡ z^0) ∨
10    (fresh [h, t, r] (
11        x^3 ≡ h^1 : t^2 ∧
12        z^0 ≡ h^1 : r^1 ∧
13        append^o t^2 y^2 r^1
14    ))
```

Рис. 22: Результат аннотирования отношения $append^o$ в обратном направлении

4.4.3. Отношение $revers^o$ в обратном направлении

Ещё один пример — отношение $revers^o$. Оно связывает два списка, получающиеся переворачиванием друг друга. Его определение приведено в листинге 23.

Добавим $revers^o$ по обратному направлению в стек вызовов и проинициализируем y как входную переменную. Рассмотрим второй дизъюнкт. На первом шаге можно попытаться проаннотировать только вызов $append^o$ в строке 20 — известна y. Такого отношения в стеке вызовов нет — добавляем и вызываем аннотирование. Это и есть вызов $append^o$

в обратном направлении, рассмотренный выше (см. рисунок 22). Аннотирование $append^o$ позволяет определить аннотации переменных r и h — распространяем их по другим конъюнктам. На следующем шаге вычисляем аннотацию переменной t рекурсивного вызова $revers^o$, так как он уже есть в стеке (см. строку 20). Распространяем аннотацию t и аннотируем x на следующем шаге в строке 18.

```
15  revers^o x^1 y^0 =
16    (x^1 ≡ [] ∧ y^0 ≡ []) ∨
17    (fresh [h, t, r] (
18         x^5 ≡ h^2 : t^4 ∧
19         append^o r^3 [h^2] y^0
20         revers^o t^4 r^3 ∧
21    ))
```

Рис. 23: Результат аннотирования отношения $revers^o$ в обратном направлении

4.5. Нормализация программ для аннотирования

4.5.1. Нерекурсивные вызовы на конструкторах

К моменту вызова аргумент-конструктор может быть проаннотирован частично. В этом случае неизвестно является ли переменная, соответствующая данному аргументу, входной или выходной. Другими словами, невозможно определить направление вызова.

Для решения данной проблемы будем действовать следующим образом:

- Сформируем новое отношение, принимающее на вход все переменные аргументов вызова. Его тело — тело вызываемого отношения с подставленными в него аргументами.
- Вызов старого отношения на аргументах-конструкторах заменим на вызов нового отношения на аргументах-переменных.

Рассмотрим вызов $append^o\ (a : as)\ ys\ z$. Один из его аргументов — конструктор списка. Сформируем новое отношение $append^o 1$ (см. рису-

нок 24), осуществив подстановку $x \rightarrow (a : as)$ в тело $append^o$. Заметим, что первый дизъюнкт $append^o$ отсутствует в $append^o1$. Он стал заведомо ошибочен: унификация $x \equiv []$ обратилась в $(a : as) \equiv []$. Во втором дизъюнкте первый конъюнкт обратился в унификацию двух конструкторов и, как следствие, разбился на две унификации.

```
41  append^o1 a as y z =
42    (fresh [h, t, r] (
43         a ≡ h ∧
44         as ≡ t ∧
45         z ≡ h : r ∧
46         append^o t y r
47    ))
```

Рис. 24: Отношение $append^o1$, полученное подстановкой $x \rightarrow (a : as)$ в $append^o$

Производить замену вызова на аргументах-конструкторах нужно также в теле созданного отношения, поэтому данный алгоритм должен запускаться до достижения неподвижной точки. В связи с этим алгоритм может зацикливаться при работе с рекурсивными вызовами на конструкторах.

4.5.2. Рекурсивные вызовы на конструкторах

Рассмотрим проблему на примере. Отношение $revacc^o$ связывает три списка: третий получается переворачиванием первого, а второй является аккумулятором. $revacc^o$ приведено на рисунке 25.

```
48  revacc^o xs acc sx =
49    (xs ≡ [] ∧ sx ≡ acc) ∨
50    (fresh [h, t] (
51         xs ≡ h : t ∧
52         revacc^o t (h % acc) sx
53    ))
```

Рис. 25: Отношение $revacc^o$

Данное отношение содержит рекурсивный вызов на конструкторе в строке 52. Попробуем заменить его на новое отношение по алгоритму,

описанному в предыдущей секции. Подстановка $a \rightarrow (h : acc)$ в $revacc^o$ представлена на рисунке 26. На данном рисунке видно, что в строке 52 такая подстановка привела к большей вложенности конструкторов. Это означает, что неподвижная точка не будет достигнута никогда.

```
48  revacc^o1 xs h acc sx =
49    (xs ≡ [] ∧ sx ≡ (h % acc)) ∨

50    (fresh [h', t] (
51        xs ≡ h' : t ∧
52        revacc^o t (h' % (h % acc))
              sx
53    ))
```

Рис. 26: Отношение $revacc^o1$, полученное подстановкой $acc \rightarrow (h : acc)$ в $revacc^o$

Альтернативное решение состоит из двух шагов:

- В дизъюнкт, содержащий рекурсивный вызов на конструкторе, добавим конъюнкт — унификацию этого конструктора с новой переменной;
- В вызове аргумент-конструктор заменим на новую переменную.

На рисунке 27 приведён пример применения данного решения и результат аннотирования $revacc^o$ в прямом направлении.

```
48  revacc^o2 xs0 acc1 sx1 =
49    (xs0 ≡ [] ∧
50    sx1 ≡ < gen :> ∧
51    sx1 ≡ acc2) ∨
52    (fresh [h, t, hacc] (
53        xs0 ≡ h1 : t1 ∧
54        hacc2 ≡ h1 : acc3 ∧
55        revacc^o t1 hacc2 sx2
56    ))
```

Рис. 27: Результат аннтирования отношения $revacc^o2$, полученного унификацией аргумента-конструктора с первым входным аргументом

Унификация позволит определять к моменту вызова, является ли аргумент, бывший конструктором, входным или выходным. “Минусом”

данного подхода является возможность потерять информацию об аннотациях переменных конструктора для аннотирования тела вызова. Потеря этой информации может привести к неуспешному завершению аннотирования. Применение алгоритма генерация способно исправить ситуацию.

Данный подход может работать и для нерекурсивных вызовов на конструкторах, но он с большей вероятностью потребует генерацию, применения которой хочется избежать.

4.5.3. Вызовы на одних и тех же переменных

Вызовы отношений могут происходить на одних и тех же переменных. В этом случае аннотации соответствующих аргументов обязаны совпадать. Это делает не валидными некоторые направления, которые, судя по количеству аргументов, должны существовать.

Рассмотрим пример: $append^o\ x\ x\ z$. У $append^o$ три аргумента и, значит, восемь направлений. Однако, первые два аргумента данного вызова совпадают и направлений остаётся четыре, так как направления с разной аннотацией первых двух аргументов становятся невалидными.

Справиться с данной проблемой помогает тот же подход, что и для нерекурсивных вызовах на конструкторах: создадим новое отношение, подставив аргументы в тело исходного. В созданном отношении (см. рисунок 28) заведомо не может существовать невалидных направлений.

```
57  append^o2 x z =
58    (x ≡ [] ∧ x ≡ z) ∨
59    (fresh [h, t, r] (
60        x ≡ h : t ∧
61        z ≡ h : r ∧
62        append^o t x r
63    ))
```

Рис. 28: $append^o 2$, полученное подстановкой $y \rightarrow x$ в $append^o$

4.6. Корректность алгоритма

Алгоритм аннотирования, представленный в работе, способен аннотировать только нормализованные программы на miniKanren. Однако, любую программу на miniKanren можно привести в нормальную форму описанными выше методами. Таким образом, доказав корректность аннотирования нормализованных программ, мы докажем и корректность ненормализованных.

Алгоритм представляет собой адаптацию алгоритм анализа времени связывания для miniKanren. Для доказательства корректности необходимо показать его терминируемость и согласованность, что и сделано в последующих подчастях.

4.6.1. Терминируемость

Алгоритм терминируется, так как повторное аннотирование отношений не производится. Имеющиеся в стеке вызовов отношения не аннотируются снова, а в каждом отношении используется конечное количество уникальных переменных. Это значит, что каждому отношению можно сопоставить конечное количество уникальных аннотаций.

Существование нескольких вызовов в одном дизъюнке приводит к необходимости применять алгоритм аннотирования ко всем возможным версиям дизъюнкта, каждая их которых отличается очередной перестановкой вызовов. Терминируемость в этом случае следует из двух фактов:

Количество перестановок вызовов конечно и, значит, конечно количество версий дизъюнкта;

Алгоритм аннотирования терминируется (доказано выше).

Добавление генерации также не повлияет на терминируемость несмотря на итеративность процесса. Количество переменных, оставшихся в случае неуспешного аннотироавния помеченными $Undef$, конечно для всего стека вызовов, так как конечно количество переменных в любом отношении, а, значит, и в стеке вызовов. На каждой итерации генерации происходит добавление хотя бы одной генерации хотя бы в одно

определение со стека вызовов.

Терминируемость доказана.

4.6.2. Согласованность

В анализе времени связывания под согласованностью понимается зависимость статических данных только от статических: статические данные не могут определяться динамическими.

Вычисление дизъюнктов в miniKanren происходит независимо, значит, и аннотировать их можно независимо. Как следствие, “проаннотировать тело отношения” означает “проаннотировать несколько дизъюнктов”. Показав корректность аннотирования одного дизъюнкта, покажем корректность аннотирования всего тела.

Каждый дизъюнкт — это конъюнкция вызовов и унификаций. Вычисление конъюнктов в miniKanren происходит одновременно: значение полученное в одном конъюнкте, мгновенно становится известно в другом. Для аннотирования это означает, если стала известна аннотация целевой переменной в одном конъюнкте, она мгновенно становится известна во всех конъюнктах, в которые эта переменная входит. Именно так и происходит в алгоритме: дизъюнкты аннотируются независимо, а аннотация переменной, ставшая известной в одном конъюнкте, распространяется на все вхождения этой переменной в другие конъюнкты.

Введём понятие зависимости одной переменной от другой в рамках предложенного алгоритма. Понятия отношения и унификации ”равноправны”, но при выборе конкретного направления вычисления значения переменных множества X неизбежно становятся известны раньше значений переменных множества Y. В этом случае будем говорить, что переменные Y *зависят* от переменных X.

Пример: зависимость для унификаций. Пусть есть два конъюнкта: $x \equiv y$ и $y \equiv 7$. Во втором конъюнкте 7 — константа, поэтому мы можем проунифицировать y и сказать, что $y = 7$. В этот же момент мы узнаем в первом конъюнкте, что y стала известна и можем превратить унификацию в равенство $x = y$. Это и назовём зависимостью x от y.

Пример: зависимость для вызовов отношений. Пусть есть вызов отношения $append^o\ x\ y\ z$, где мы уже знаем из других конъюнктов значение z. В этом случае алгоритм посчитает, что этот вызов $append^o$ происходит в обратном направлении и переменные x и y являются выходными. В этом случае можно говорить о зависимости x и y от z: $(x,\ y)\ =\ append^o\ z$ (в случае недетерминированной семантики $apppend^o$ корректнее говорить о $[(x,\ y)]\ =\ append^o\ z$).

Введём инвариант, отражающий идею согласованности. Доказав его выполнение на любом шаге алгоритма, мы докажем его корректность.

Инвариант:

- *В любой момент времени переменная может быть не проаннотирована (иметь аннотацию $Undef$);*
- *Если переменная проаннотирована числом, то существует хотя бы один конъюнкт, в котором все переменные, от которых она зависит, проаннотированы строго меньшими числами;*

Рассмотрим алгоритм ещё раз, чтобы убедиться в выполнении инварианта. В начальный момент времени числовую аннотацию 0 имеют только входные переменные. Остальные переменные проаннотированы $Undef$.

Конъюнкты отсортированы: вызовы следуют за унификациями. Последовательно обходим все унификации. К каждой применяется алгоритм аннотирования унификаций, в точности выполняющий инвариант. $Undef$-аннотация целевой переменной заменяется всегда на строго большее значение, чем значение аннотации любой переменной, от которой целевая переменная зависит. После аннотирования каждого конъюнкта информация об аннотациях его переменных распространяется на все оставшиеся конъюнкты. Следующий для аннотирования конъюнкт обладает релевантными аннотациями.

При таком подходе к моменту необходимости аннотировать первый вызов отношения мы можем быть уверены, что в текущем вызове известны все аннотации переменных, которые можно было получить из

унификаций. Все другие — только из последующих вызовов отношений. Тем самым, мы знаем направление первого вызова. При наличии нескольких вызовов их порядок влияет на аннотирование. Наилучший порядок, позволяющий получить проаннотированное отношение, можно найти только опытным путём — перебрав все перестановки вызовов. Поэтому, без ограничения общности можно считать, что первый вызов выбран верно. Если аннотирование при этом закончится неудачей, запустится аннотирование того же дизъюнкта с другим порядком вызовов. Важно заметить, что, в случае неуспеха аннотирования стек вызовов будет содержать переменные с $Undef$ аннотациями — это является частью инварианта.

Вернёмся к аннотированию вызова. Алгоритм аннотации аргументов вызова в точности соблюдает инвариант. Каждое вызываемое в конкретном направлении отношение добавляется в стек, если оно там отсутствовало, и инициализируется так, что его входные переменные имеют аннотацию 0. Это позволяет рассматривать аннотацию тела вызываемого отношения независимо от причин аннотирования: является ли аннотируемая цель целью программы или телом вызываемого отношения.

Тем самым согласованность доказана.

5. Тестирование

В первой части данного раздела предствлено несколько классификаций программ на miniKanren, для классов каждой из которых указана возможность трансляции. Описание системы тестирования полученного алгоритма трансляции находится во второй части.

5.1. Классификации программ для трансляции и ограничения подхода

В разделе описаны четыре классификации программ на miniKanren. Для каждой из классификаций указаны виды программ, не поддерживаемые транслятором.

5.1.1. $In - Out$ классификация

Программы на miniKanren, отправляемые на трансляцию, можно разделить по направлениям трансляции. Отсюда можно выделить следующие виды:

- Все аргументы являются выходными;
- Все аргументы являются входными;
- Часть аргументов — входные, часть — выходные.

Транслятор не поддерживает только первый тип. Он отклоняется процессе аннотирования вызова: раскрытия не происходит, если все аннотации его аргументов — $Undef$, так как недостаточно информации для аннотирования.

5.1.2. Классификация для аннотатора

В процесс разработки алгоритма аннотирования происходил итеративно. На первой итерации был разработан алгоритм для нормализованных программ на miniKanren. На последующих — постепенно до-

бавлялись оставшиеся конструкции miniKanren и свойства. Естественно произвести классификацию программ в соответствии с использованием тех или иных конструкций и свойств. Одна и та же программа может содержать разные конструкции, и, как следствие, находиться одновременно в нескольких классах.

- Тело отношения находится в ДНФ;
- $fresh$-переменные только на верхнем уровне;
- Присутствуют унификации двух конструкторов;
- Ни в каком дизъюнкте нет вызовов (только унификации);
- В каждом дизъюнкте не более одного вызова на аргументах-переменных;
- Несколько вызовов на аргументах-переменных;
- Унификация $fresh$-переменных только друг с другом;
- Вызовы на аргументах-конструкторах;
- Вызовы на одних и тех же переменных.

Все перечисленные виды поддерживаются.

5.1.3. Классификация по особенностям miniKanren

Как и в классификации для аннотатора, программа принадлежит конкретному классу, если содержит соответствующую конструкцию или отвечает определённому свойству. В отличие от классификации для аннотатора, данная классификация принимает во внимание не только саму программу на miniKanren, но и текущее направление трансляции. Направление вычисления влияет на проявление особенностей. Например, при одном направлении перекрытие дизъюнктов не сможет произойти, так как каждый дизъюнкт рассматривает один из конструкторов входной переменной. Поменяв направление, получим другой набор входных переменных, для которых соответствующих унификаций, перебирающих конструкторы, может уже не быть.

- Несколько выходных переменных;
- Пересечение результатов дизъюнктов;
- Недетерминированность результатов;
- Унификация $fresh$-переменных только друг с другом.

Все перечисленные виды поддерживаются.

5.1.4. Классификация для транслятора

Перечисляет проблемы, возникающие при трансляции. Каждая проблема разрешается добавлением в итоговую программу новой конструкции. Таким образом, эта классификация также распределяет транслированные программы по наличию тех или иных конструкций. Одна программа может принадлежать нескольким классам.

- Совпадение имён в сопоставлениях с образцом (охранные выражения);
- Совпадение имён в определениях (условие для ветвления);
- Конструкторы и генерация (запрашивается у пользователя);
- Несколько выходных переменных при вызове (возврат результата в кортеже);
- Определение возникло из унификации (выражение вида $let\ x\ =\ y$);
- Определение возникло из вызова (выражение вида $x \leftarrow func\ y$);

Классификация не содержит проблемы, которые привели к постоянному использованию определённых конструкций. Пример — проблема перекрытия результатов дизъюнктов, так как теперь каждая функция является конкатенацией результатов вспомогательных функций.

Все перечисленные виды поддерживаются.

5.2. Тестирование

Данная часть описывает способ тестирования разработанного алгоритма трансляции.

Он принимает на вход программу в абстрактном синтаксисе miniKanren и выдаёт программу в абстрактном синтаксисе функционального языка. Такой результат алгоритма невозможно запустить. В то же время наилучшим способом тестирования является именно запуск полученной функции. Это влечёт за собой необходимость создать транслятор абстрактного синтаксиса функционального языка в конкретный. Кроме того, работать с miniKanren в абстрактном синтаксисе так же не очень удобно. В связи с чем разработан простейший конкретный синтаксис miniKanren и реализован его парсер.

5.2.1. Парсер конкретного синтаксиса miniKanren

miniKanren — встраиваемый язык и это одно из его преимуществ, поэтому при разработке конкретного синтаксиса не стояло цели создать идеальный. Цель его создания — удобства тестирования.

Была создана соответствующая грамматика (см. приложение) и написан парсер на Haskell.

Пример конкретного синтаксиса приведён на рисунке 29. В строке 1 присутствует слово *conde* — оно является синтаксическим сахаром для дизъюнкции нескольких дизъюнктов.

```
1  :: appendo x y xy = conde
2     (x === [] /\ xy === y)
3     ([h t r:
4         x === h % t /\
5         xy === h % r /\
6         {appendo t y r}])
```

Рис. 29: Пример конкретного синтаксиса miniKanren

5.2.2. Транслятор абстрактного синтаксиса функционального языка в конкретный

Алгоритм трансляции абстрактного синтаксиса функционального языка в конкретный тривиален. Он представляет собой последовательный обход всех конструкций абстрактного синтаксиса функционального языка и печать их форме, соответствующей конкретному синтаксису Haskell.

Заметим, что можно выбрать конкретный синтаксис любого другого функционального языка программирования.

5.2.3. Система тестирования

Для тестиравания была создана база программ на miniKanren, покрывающая все варианты классификаций. Само тестирование выглядит так:

- Запуск парсера на программа в конкретном синтаксисе miniKanren;
- Трансляция в абстрактный функцильнаый синтаксис;
- Трансляция в Haskell;
- Запуск *unit*-тестов на транслированной Haskell-программе с целью проверить, что программа обладает желаемым поведением.

По результатам *unit*-тестирования можно утверждать, что данный алгоритм работает для всех выделенных типов программ miniKanren; ограничением трансляции является невозможность трансляции отношений на направлении, когда все аргументы являются выходными.

Заключение

Целью данной работы было создание транслятора реляционного языка в функциональный, способного в транслированной функции сохранить семантику исходного отношения в выбранном направлении.

Для достижения этой цели было поставлено несколько задач, каждая из которых была решена.

- Разработка алгоритма транслирования абстрактного синтаксиса реляционного языка в абстрактный синтаксис функционального языка.

  Рассмотрены особенности miniKanren и особенности трансляции. С их учётом создан абстрактный синтаксис функционального языка и разработан алгоритм трансляции. Реализация написана на Haskell. Полученный алгоритм способен транслировать программы на miniKanren во всех направлениях за исключением, когда все переменные являются выходными. Доказано сохранение семантики при трансляции.

- Разработка алгоритма аннотирования, позволяющего транслятору определять направления и порядок вычислений конъюнктов.

  Для программ на miniKanren введено понятие нормальной формы. На основе идеи анализа времени связывания разработан алгоритм аннотирования переменных для нормализованных программ. Рассмотрены способы приведения любых программ в нормальную форму как часть алгоритма аннотирования. Его реализация написана на Haskell. Доказана корректность.

- Тестирование и анализ результатов.

  Предложено несколько классификаций программ на miniKanren в соответствии с проблемами, возникшими при создании алгоритмов аннотирования и трансляции. Создана база программ на miniKanren, покрывающая предложенные классификации. Для

тестирования результата трансляции в конкретном синтаксисе создан конкретный синтаксис miniKanren (грамматика в приложении) и реализован его парсер, а так же транслятор абстрактного функционального синтаксиса в конкретный. Оба алгоритма реализованы на Haskell.

В рамках данной работы получены следующие результаты:

- Разработан алгоритм трансляции реляционных программ в функциональные для конкретного направления с сохранением семантики транслируемых отношений;
- По результатам тестирования можно утверждать, что данный алгоритм работает для всех выделенных типов программ miniKanren; ограничением трансляции является невозможность трансляции отношений на направлении, когда все аргументы являются выходными;
- Исходный код проекта можно найти на сайте **??**, автор принимал участи под учётной записью *Pluralia*;
- Результаты работы опубликованы в сборнике конференции SEIM'20 и приняты на конференцию TEASE-LP'20.

Список литературы

[1] Bellia Marco, Levi Giorgio. The relation between logic and functional languages: a survey // The Journal of Logic Programming. — 1986. — Vol. 3, no. 3. — P. 217 – 236.

[2] Hemann Jason, Friedman Daniel P. uKanren: A Minimal Functional Core for Relational Programming. — 2013.

[3] Jones Neil D, Gomard Carsten K, Sestoft Peter. Partial evaluation and automatic program generation. — Peter Sestoft, 1993.

[4] Lozov Petr, Verbitskaia Ekaterina, Boulytchev Dmitry. Relational Interpreters for Search Problems // Relational Programming Workshop. — 2019. — P. 43.

[5] Marchiori Massimo. The functional side of logic programming // FPCA. — 1995. — P. 55–65.

[6] Matsushita Tatsuru, Runciman Colin. Functional Counterparts of some Logic Programming Techniques. — 1997.

[7] Somogyi Zoltan, Henderson Fergus, Conway Thomas. The execution algorithm of mercury, an efficient purely declarative logic programming language // The Journal of Logic Programming. — 1996. — Vol. 29, no. 1. — P. 17 – 64.

[8] Specialising Interpreters Using Offline Partial Deduction / Michael Leuschel, Stephen-John Craig, Maurice Bruynooghe, Wim Vanhoof. — Vol. 3049. — 2004. — 01. — P. 340–375.

[9] Specialising Interpreters Using Offline Partial Deduction / Michael Leuschel, Stephen-John Craig, Maurice Bruynooghe, Wim Vanhoof. — Vol. 3049. — 2004. — 01. — P. 340–375.

[10] Thiemann Peter. A Unified Framework for Binding-Time Analysis // TAPSOFT. — 1997.

[11] A Unified Approach to Solving Seven Programming Problems (Functional Pearl) / William E. Byrd, Usamichael Ballantyne, Usagregory Rosenblatt, Matthew Might // Relational Programming Workshop. — 2017.

[12] Vanhoof Wim, Bruynooghe Maurice, Leuschel Michael. Binding-time analysis for Mercury // Program Development in Computational Logic. — Springer, 2004. — P. 189–232.

A. Грамматика языка miniKanren

⟨*program*⟩ ::= ⟨*def*⟩* ⟨*goal*⟩

⟨*term*⟩ ::= ⟨*ident*⟩ | ‘<’ ⟨*ident*⟩ ‘:’ ⟨*term*⟩* ‘>’

⟨*def*⟩ ::= ‘::’ ⟨*ident*⟩ ⟨*ident*⟩* ‘=’ ⟨*goal*⟩

⟨*goal*⟩ ::= ⟨*disj*⟩ | ⟨*fresh*⟩ | ⟨*invoke*⟩

⟨*fresh*⟩ ::= ‘[’ ⟨*ident*⟩+ ‘:’ ⟨*goal*⟩ ‘]’

⟨*invoke*⟩ ::= ‘{’ ⟨*ident*⟩ ⟨*term*⟩* ‘}’

⟨*disj*⟩ ::= ⟨*conj*⟩ (‘\/’ ⟨*conj*⟩)*

⟨*conj*⟩ ::= ⟨*pat*⟩ (‘/\’ ⟨*pat*⟩)*

⟨*pat*⟩ ::= ⟨*term*⟩ ‘===’ ⟨*term*⟩ | ⟨*fin*⟩

⟨*fin*⟩ ::= ⟨*fresh*⟩ | ‘(’ ⟨*disj*⟩ ‘)’