

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.

Уточните, кто у
вас научник
формально

Научный руководитель:
Вербицкая Е. А.

Рецензент:
Березун Д. А.

Уточните,
нужны ли
регалии. Даня --
кандидат
каких-то наук.

Санкт-Петербург
2020

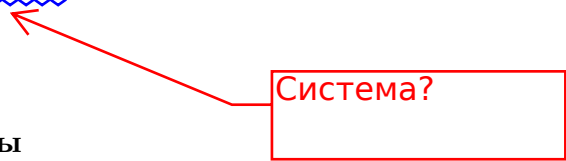
Оглавление

Введение	4
1. Обзор предметной области и постановка задачи	7
1.1. Язык программирования miniKanren	7
1.2. Трансляция в функциональный язык	8
1.2.1. Альтернативные подходы к ускорению программ	8
1.2.2. Основная проблема трансляции	8
1.2.3. Обзор существующих решений	10
1.3. Анализ времени связывания	10
1.3.1. Обзор существующих решений	10
1.4. Цель и задачи работы	11
2. Разработка аннотатора	12
2.1. Анализ времени связывания для miniKanren	12
2.2. Нормальная форма программы на miniKanren	13
2.3. Алгоритм аннотирования нормализованной программы	14
2.4. Примеры аннотирования	16
2.4.1. <i>append</i> ^o в прямом направлении	16
2.4.2. <i>append</i> ^o в обратном направлении	17
2.4.3. <i>revers</i> ^o в обратном направлении	17
2.5. Расширение множества аннотируемых программ	18
2.5.1. Несколько вызовов в одном дизъюнкте	18
2.5.2. Унификация fresh-переменных только друг с другом	20
2.5.3. Нерекурсивные вызовы на конструкторах	21
2.5.4. Рекурсивные вызовы на конструкторах	22
2.5.5. Вызовы на одних и тех же переменных	23
2.6. Корректность алгоритма	24
2.6.1. Терминируемость	24
2.6.2. Согласованность	25
3. Разработка транслятора	27
3.1. Особенности miniKanren и способы их трансляции	27
3.1.1. Несколько выходных переменных	27
3.1.2. Перекрывание результатов дизъюнктов	28
3.1.3. Недетерминированность результатов	28
3.1.4. Порядок и направление при исполнении отношений	29
3.1.5. Переменные, принимающие все возможные значения	30
3.1.6. <u>Цикл только за счёт рекурсии</u>	30

пересечение,
наверное

Что?

3.2. Особенности трансляции	31
3.2.1. Сопоставление с образцом для входных переменных	31
3.2.2. Перекрытие имён в сопоставлениях с образцом	32
3.2.3. Перекрытие имён в определениях	33
3.2.4. Трансляция конструкторов	33
3.3. Алгоритм трансляции	34
3.3.1. Абстрактный синтаксис функционального языка	34
3.3.2. Алгоритм трансляции	35
3.4. Корректность алгоритма	37
4. Тестирование и анализ результатов	39
4.1. Классификации программ для трансляции и ограничения подхода . . .	39
4.1.1. <i>In – Out</i> классификация	39
4.1.2. Классификация для аннотатора	39
4.1.3. Классификация по особенностям miniKanren	40
4.1.4. Классификация для транслятора	40
4.2. Тестирование	41
4.2.1. Парсер конкретного синтаксиса miniKanren	41
4.2.2. Транслятор абстрактного синтаксиса функционального языка в конкретный	42
4.2.3. <u>Алгоритм тестирования</u>	42
Заключение	43
Список литературы	44
Приложение А. Грамматика языка miniKanren	45



Введение

буллет-поинты
тут добавят
объема
нахаляву

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 искомым аргумент будем обозначать знаком “?”). Выполнение отношения $add^o\ x\ y\ ?$ с зафиксированными (входными) первым и вторым аргументом найдет их сумму, а $add^o\ ?\ y\ z$ найдет такие числа, которые в сумме с y дадут z . Также можно найти одновременно значения нескольких аргументов: $add^o\ ?\ ?\ z$ найдет такие пары чисел, что в сумме они равны z , а $add^o\ ?\ ?\ ?$ 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 Mercury¹ [7]. Основным представителем парадигмы реляционного программирования является семейство интерпретируемых языков miniKanren². Языки семейства miniKanren компактны и встраиваются в языки общего назначения, за счёт чего их 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 [10]; 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 некото-

¹Официальный сайт языка Mercury: <https://mercurylang.org/>, дата последнего посещения: 14.05.2020

²Официальный сайт языка miniKanren: <http://minikanren.org/>, дата последнего посещения: 14.05.2020

ром 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.

К сожалению, выполнение отношения в обратном направлении обычно крайне не эффективно. В данной работе представлен подход улучшения производительности программы в заданном направлении. Для этого по отношению с фиксированным направлением генерируется функция на функциональном языке. В качестве конкретного функционального языка выбран Haskell.

Кратко о следующих главах

- В Главе 1 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 формулируется цель данной работы, а также определяются задачи, решаемые в последующих главах.
- В главе 2 описывается разработка алгоритма аннотирования на основе анализа времени связывания. Вводится понятие нормализованной программы, на которой работает алгоритм аннотирования. Доказывается корректность предложенного алгоритма.
- Глава 3 начинается с разбора особенностей трансляции. Далее приводится сам алгоритм трансляции. В конце доказывается корректность предложенного алгоритма.
- Глава 4 посвящена анализу результатов работы транслятора. В начале описывается способ его тестирования, включающий в себя написание парсера конкретного синтаксиса miniKanren, а так же транслятор абстрактного функционального языка в конкретный. Затем вводятся несколько классификаций программ на miniKanren и анализируются ограничения предложенного алгоритма трансляции.

Список терминов и сокращений

Термины

interleaving парсер конъюнкт дизъюнкт

?

Тут нужна мотивация работы: для ускорения есть специализация, но она (если это не Джонс-оптимальный частичный вычислитель) не позволяет избавиться от всего интерпретационного оверхеда (надо поискать, как оверхед по-русски). Джонс-оптимальный частичный вычислитель для пролога есть, для миниканрена -- нет. Мы решили попробовать альтернативный подход с трансляцией. Тут нужны ссылки, посмотрите их в работе про реляционные интерпретаторы

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. Обзор предметной области и постановка задачи

1.1. Язык программирования miniKanren

Технически это неправда: в миниканре не есть дополнительные конструкции, которые нереляционны. Ядро (или микроканрен), при этом, чистое. Надо сразу сказать, что несмотря на все это.

Семейство языков miniKanren дало рождение парадигме реляционного программирования. Это минималистичные языки, встраиваемые в языки программирования общего назначения. Как и Prolog они обладают свойством *полиmodalности* — 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: 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 *fresh*. Абстрактный синтаксис языка приведен ниже:

$$\begin{aligned} \text{Goal} : & \text{Goal} \vee \text{Goal} && (\text{дизь}) \\ & | \text{Goal} \wedge \text{Goal} && (\text{конь}) \\ & | \text{Term} \equiv \text{Term} && (\text{униф}) \\ & | \text{call Name [Term]} \\ & | \text{fresh [Var] Goal} \\ \text{Term} : & \text{Var} \\ & | \text{cons Name [Term]} \end{aligned}$$

Пример программы на языке miniKanren, связывающей три списка, где третий является конкатенацией первых двух, приведен на рисунке **??**. Для краткости [] заменяет пустой список (*cons Nil []*); $h : t$ обозначает список с головой h и хвостом t (*cons Cons [h, t]*), а $[x_0, x_1, \dots, x_n]$ — список с элементами x_0, x_1, \dots, x_n . Вызов отношения *call relation [t₀, ..., t_k]* записывается как *relation t₀ ... t_k*.

Исполнение этого отношения в прямом направлении на двух заданных списках

```

1  append° x y z =
2    (x ≡ [] ∧ y ≡ z) ∨
3    (fresh [h, t, r] (
4      x ≡ h : t ∧
5
6      z ≡ h : r ∧
7
8      append° t y r
9    ))

```

Тут еще можно
добавить про то,
что миниканрен
не всегда
перечисляет все
результаты, а
иногда
использует
метапеременны
е

Рис. 1: Пример программы на miniKanren

$append^{\circ} [1, 2] [3]$? 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 $append^{\circ} ? ? [1, 2, 3]$ является множество пар $\{([], [1, 2, 3]), ([1], [2, 3]), ([1, 2], [3]), ([1, 2, 3], [])\}$.

1.2. Трансляция в функциональный язык

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

Это не
подглава
трансляции.
Убрать
отсюда

Утверждается, что транслированная версия отношения будет выполняться быстрее, чем оригинальная в направлении трансляции. Преимущество заключается уже в том, что встраиваемый miniKanren имеет затраты на интерпретацию в хостовый язык, в то время как транслированная версия будет выполняться на хостовом языке сразу же.

Кем?

1.2.1. Альтернативные подходы к ускорению программ

В [4] для решения этой проблемы используется специализация. В статье показано, что специализация приводит к существенному приросту скорости работы программы. Однако, чтобы избавиться от всех накладных расходов, связанных с интерпретацией программы, необходим Джонс-оптимальный специализатор [3], реализация которого — нетривиальная задача. Трансляция позволяет избежать затрат на интерпретацию.

~~1.2.2. Основная проблема трансляции~~

Особенностью реляционного программирования является отсутствие строго 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,

Убрать
заголовок, это
все лучше будет
смотреться
сплошным
текстом.

это же
объясн

а так же их порядок исполнения с учётом направления трансляции. Для решения такой задачи используется *анализ времени связывания* (binding time analysis).

Отношение, выполненное в заданном направлении, можно рассматривать как функцию из известных аргументов в неизвестные. Например, отношение $append^o$, выполненное в прямом направлении ($append^o\ x\ y\ ?$), 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 листинге 2. В двух уравнениях первая переменная 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. 2: Результат трансляции $append^o\ x\ y\ ?$

Пример трансляции $append^o\ ?\ ?\ z$ приведен в листинге 3.

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

```

13      appendo00I x0 = appendo00I0 x0 ++ appendo00I1
        x0
14      appendo00I0 s4@s0 = do
15        let s3 = []

16        return $ (s3, s0)
17      appendo00I0 _ = []

18      appendo00I1 s4@(s5 : s7) = do
19        (s6, s0) <- appendo00I s7

20        let s3 = (s5 : s6)

21        return $ (s3, s0)
22      appendo00I1 _ = []

```

Рис. 3: Результат трансляции *append* ? ? z

1.2.3. Обзор существующих решений

*Вот это обзор
данных*

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.

1.3. Анализ времени связывания

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.

1.3.1. Обзор существующих решений

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

Анализ времени связывания существует для логического языка Prolog [8] и функционально логического языка Mercury [11] — представителей родственных реляционному программированию парадигм. Однако, в языке Mercury анализ времени связывания [11] 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 [8]. Основное предназначение анализа в этой работе — улучшение качества специализации, упорядочивания вызовов не производится.

Работа [9] 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 $\{0, 1, \dots, N\}$. Эта идея была использована в данной работе.

← Это отдельный раздел обычно, начинающийся на отдельной странице

1.4. 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 транслирования ~~абстрактного синтаксиса~~ реляционного языка в ~~абстрактный синтаксис~~ функционального языка.
- Тестирование разработанного инструмента и анализ результатов.

Это жаргон, по
стилю не
подходит.
Анализ времени
связывания или
алгоритм
аннотации
звучит лучше

2. Разработка аннотатора

До аннотаций
должна быть
глава про
трансляцию,
которая чего-то
забыла в обзоре

Данная глава посвящена разработке аннотатора. В первой части рассказано об адаптации идей анализа времени связывания для определения направления вычислений. Вторая часть вводит понятие нормальной формы для программы на miniKanren. Алгоритм аннотирования описывается в третьей части. В четвёртой части представлена корректность предложенного алгоритма.

2.1. 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.

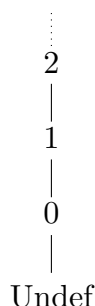


Рис. 4: Полурешетка на аннотациях

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

Это-то зачем?
можно было
унифицировать
всегда $Var ==$
 $Term$ -- вне
зависимости от
положения
переменной
унификация
работает
одинаково

Рис. 5: Абстрактный синтаксис нормализованной программы на miniKanren

2.2. Нормальная форма программы на miniKanren

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 5.

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

- Тело определения находится в ДНФ;
- *fresh*-цель одна на самом верхнем уровне;
- Не существует унификаций термов-конструкторов;
- Не существует вызовов на термах-конструкторах;

Все свободные
переменные
введены при
помощи *fresh* на
самом верхнем
уровне

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

Жаргонизм:
область
видимости

- ДНФ тела позволяет уменьшить глубину вложенности программы;
- *fresh*-цель задаёт скоуп вычислений и позволяет использовать одинаковые имена переменных в различных скоупах — её наличие только на верхнем уровне означает, что все переменные находятся в одном скоупе и на всю программу существует только один скоуп;
- Отсутствие унификаций термов-конструкторов позволяет не производить очевидной унификации в процессе выполнения алгоритма;
- Отсутствие вызовов на термах-конструкторах позволяет избежать неопределённости в процессе аннотирования;

Не было этого термина раньше

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

- Приведение булевого выражения в ДНФ — 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 подчастях описания алгоритма;

Подглавах/подразделах

Не текст: добавить связующие предложения

2.3. Алгоритм аннотирования нормализованной программы

Входные данные алгоритма: нормализованная программа на miniKanren (цель и список определений) и список входных переменных. Выходные данные: 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 ~~нового аннотирования не произошло~~, считается, что достигнута неподвижная точка. Для аннотации цели в дизъюнктивной 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

не одна ли переменная в унификации или вызове?

Ни одна новая переменная не была проаннотирована

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 $x^{Undef} \equiv t[y_0^{i_0}, \dots, y_k^{i_k}]$, 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 $n = \max\{i_0, \dots, i_k\}$;
- Переменная, аннотированная числом, унифицируется с термом: $x^n \equiv t[y_0^{i_0}, \dots, y_k^{i_k}]$; 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.

Кавычки в теке:

`` слева другие, чем справа "

Справа два амперсанда, это не двойная кавычка

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. При этом, 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.

2.4. Примеры аннотирования

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

Отношение
appendo

2.4.1 *appendo* в прямом направлении

Любое англоязычное слово должно быть
сопровождено русским
определяющим

Аннотирование отношения *appendo* в прямом направлении представлено на рисунке 6. *appendo* — отношение связывающее три списка, первые два из которых являются конкатенацией третьего. 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 *appendo*. Так как x и y — входные переменные, их аннотации нам известны. Аннотация первого дизъюнкта тривиальна, поэтому рассмотрим второй. 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

Лучше рассмотрите, никогда не стоит
недооценивать, насколько человеку не в теме
нетривиально тривиальное

имеющимся в стеке, поэтому можно проаннотировать переменную r . Распространяем аннотацию r в строке 5. На последнем шаге аннотируем z в строке 4.

```

1  appendo x0 y0 z1 =
2    (x0 ≡ [] ∧ y0 ≡ z1) ∨
3    (fresh [h, t, r] (
4      x0 ≡ h1 : t1 ∧
5      z3 ≡ h1 : r2 ∧
6      appendo t1 y0 r2
7    ))

```

Рис. 6: Аннотирование append^o в прямом направлении

2.4.2. append^o в обратном направлении

Теперь рассмотрим аннотирование *append^o* в обратном направлении. В этом случае мы считаем переменную z входной (см. рисунок 19). Пусть *append^o* уже в стеке и z проаннотирована. В первом дизъюнкте 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 13). Аннотация r распространяется на строку 14. Отношение с согласованным направлением есть в стеке, поэтому можно аннотировать t и y . Далее аннотация t распространяется на строку 12, и на последнем шаге аннотируется x .

```

8  appendo x1 y1 z0 =
9    (x1 ≡ [] ∧ y1 ≡ z0) ∨
10   (fresh [h, t, r] (
11     x3 ≡ h1 : t2 ∧
12     z0 ≡ h1 : r1 ∧
13     appendo t2 y2 r1
14   ))

```

Рис. 7: Аннотирование append^o в обратном направлении

2.4.3. revers^o в обратном направлении

Ещё один пример — отношение *revers^o*. Оно связывает два списка, получающиеся переворачиванием друг друга. Его определение приведено в листинге 8.

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 19).

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  reverso x1 y0 =
16    (x1 ≡ [] ∧ y0 ≡ []) ∨
17    (fresh [h, t, r] (
18      x5 ≡ h2 : t4 ∧
19      appendo r3 [h2] y0
20      reverso t4 r3 ∧
21    ))

```

Рис. 8: Аннотирование $revers^o$ в обратном направлении

2.5. Расширение множества аннотируемых программ

Каждая часть данного раздела описывает способ расширения множества программ, которые возможно проаннотировать и, в дальнейшем, транслировать.

2.5.1. Несколько вызовов в одном дизъюнкте

Существуют отношения, точная аннотация которых не возможна без вмешательства человека или полного перебора возникающих вариантов. Один из видов таких отношений — отношения, содержащие несколько вызовов в одном дизъюнкте. Пример такого отношения приведен на рисунке 9. 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).

```

22  relo x y z =
23    fo x y ∨
24    ho z y ∨
25    go x z

```

Рис. 9: Пример программы на miniKanren с несколькими вызовами в одном дизъюнкте

Рассмотрим конкретный пример. Возьмём то же определение $revers^o$, что было рассмотрено на рисунке 8, и попробуем проаннотировать его в прямом направлении (см. рисунок 10).

Во втором дизъюнкте определим аннотации переменных h и t в строке 29 и распространим их на последующие вызовы. Далее попытаемся проаннотировать тело

append^o с учётом, что только второй аргумент входной. Как показано на рисунке 11, попытка этого потерпит неудачу. Во втором дизъюнкте *append^o* при таком направлении не существует возможности узнать значение (а, значит, и время связывания) переменных *h*, *x* и *z* (данная проблема обсуждается в следующей подчасти). Мы можем успешно завершить аннотирование *revers^o*, однако, при трансляции не сможем получить работающую программу, так как не успешно завершилось аннотирование *append^o*.

```

26  reverso x0 y1 =
27    (x0 ≡ [] ∧ y1 ≡ []) ∨
28    (fresh [h, t, r] (
29      x0 ≡ h1 : t1 ∧
30      appendo r2 [h1] y2
31      reverso t1 r2 ∧
32    ))

```

Рис. 10: Аннотирование *revers^o* в прямом направлении с порядком вызовов *append^o-revers^o*

```

33  appendo x1 y0 z1 =
34    (x1 ≡ [] ∧ y1 ≡ z0) ∨
35    (fresh [h, t, r] (
36      xUndef ≡ hUndef : t1 ∧
37      zUndef ≡ hUndef : r1 ∧
38      appendo t1 y0 r1
39    ))

```

Рис. 11: Аннотирование *append^o* со вторым входным аргументом

Посмотрим, как будет происходить аннотирование, если поменять местами вызовы *append^o* и *revers^o* 12. 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 19) в строке 45.

```

40  reverso x0 y1 =
41    (x0 ≡ [] ∧ y1 ≡ []) ∨
42    (fresh [h, t, r] (
43      x0 ≡ h1 : t1 ∧
44      reverso t4 r3 ∧
45      appendo r3 [h2] y0
46    ))

```

Рис. 12: Аннотирование *revers^o* в прямом направлении с порядком вызовов *revers^o-append^o*

Таким образом, для решения проблемы нескольких вызовов в одном дизъюнкте было 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 будем запускать аннотирование дизъюнкта на каждой из версий до тех пор, пока либо аннотирование закончится успехом, либо переберём все возможные версии. В последнем случае считается, что аннотирование неуспешно.

2.5.2. Унификация fresh-переменных только друг с другом

Причина, по которой невозможно проаннотировать *append*^o со вторым входным аргументом (см. рисунок 11) — 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 симитировать данный подход, добавив генерацию переменных оставшихся свободными. Под *добавлением генерации* понимается добавление в дизъюнкт нового конъюнкта — унификации целевой переменной со списком всех допустимых значений. В этом случае аннотация такой переменной будет являться аннотацией константы и равна 1.

Реально же нет списка всех допустимых значений, это просто некоторая метка же

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, как? какая из частей унификации является подвыражением другой и будем генерировать переменные части-подвыражения.

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 $append^o$ со вторым входным аргументом с добавлением генерации приведено на рисунке ??.

```

33   $append^o\ x^1\ y^0\ z^1 =$ 
34     $(x^1 \equiv [] \wedge y^1 \equiv z^0) \vee$ 
35     $(fresh\ [h, t, r]\ ($ 
36       $h^1 \equiv <gen> \wedge$ 
37       $x^2 \equiv h^1 : t^1 \wedge$ 
38       $z^2 \equiv h^1 : r^1 \wedge$ 
39       $append^o\ t^1\ y^0\ r^1$ 
40     $))$ 

```

Рис. 13: Аннотирование $append^o$ со вторым входным аргументом с добавлением генерации

Генерация позволила проаннотировать $append^o$ со вторым входным аргументом. В этом случае аннотирование $revers^o$ с последовательностью вызовов $append^o-revers^o$ (рисунок 10) становится успешным. Это даёт два способа трансляции $revers^o$ в прямом направлении. Для примера на рисунке 10 из вызовов будут сгенерированы две функции: предикат $revers^o$ и $append^o$ со вторым входным аргументом. Для примера на рисунке 12) — 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 поместим алгоритм генерации после алгоритма перебора перестановок конъюнктов. На примере $revers^o$: из двух вариантов предпочтительной последовательностью конъюнктов обладает вариант на рисунке 12. Аннотирование второго дизъюнкта на рисунке 10 завершится неудачей, поэтому произойдёт перестановка вызовов и получим последовательность конъюнктов другого варианта. Это приведёт к успеху аннотирования и генерация не понадобится.

Вызов разве что. Алгоритм это абстракция, ее нельзя поставить вперед другой абстракции

2.5.3. 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 пример. Пусть существует вызов на аргументе конструкторе $append^o$ ($a : as$) $ys\ z$. Сформируем новое отношение $append^o1$ (см. рисунок 14), 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       $a \equiv h \wedge$ 
44       $as \equiv t \wedge$ 
45       $z \equiv h : r \wedge$ 
46       $append^o\ t\ y\ r$ 
47     $))$ 

```

Рис. 14: $append^o1$, полученное подстановкой $x \rightarrow (a : as)$ в $append^o$

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

так же = таким же образом;
также = и

2.5.4. Рекурсивные вызовы на конструкторах

Рассмотрим проблему на примере. Отношение $revacc^o$ связывает три переменные: третья получается переворачиванием первой, а вторая является аккумулятором. $revacc^o$ приведено на рисунке 15.

```

48   $revacc^o\ xs\ acc\ sx =$ 
49     $(xs \equiv [] \wedge sx \equiv acc) \vee$ 
50     $(fresh\ [h,\ t]\ ($ 
51       $xs \equiv h : t \wedge$ 
52       $revacc^o\ t\ (h \% acc)\ sx$ 
53     $))$ 

```

шрифт другой

Рис. 15: 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 16. 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  revacco1 xs h acc sx =
49    (xs ≡ [] ∧ sx ≡ (h % acc)) ∨
50    (fresh [h', t] (
51      xs ≡ h' : t ∧
52      revacco t (h' % (h % acc))
53      sx
54    ))

```

Рис. 16: $revacc^o1$, 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 17 приведён пример применения данного решения и аннотирование $revacc^o$ с первым входным аргументом.

```

48  revacco2 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      revacco t1 hacc2 sx2
56    ))

```

Рис. 17: $revacc^o$, полученное унификацией аргумента-конструктора, с первым входным аргументом

Унификация позволит определять к моменту вызова, является ли аргумент, бывший конструктором, входным или выходным. Однако, так может потеряться информация об аннотации переменных, входящих в состав аргумента-конструктора. Это может препятствовать успешному аннотированию. Применение генерации поможет с этим справиться.

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

2.5.5. 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 ва-

лидными 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 вызовов на конструкторах: создадим новое отношение, подставив аргументы в тело исходного. В созданном отношении (см. рисунок 18) заведомо не может существовать невалидных направлений.

```

57   $append^o2\ x\ z =$ 
58     $(x \equiv [] \wedge x \equiv z) \vee$ 
59     $(fresh\ [h, t, r]\ ($ 
60       $x \equiv h : t \wedge$ 
61       $z \equiv h : r \wedge$ 
62       $append^o\ t\ x\ r$ 
63     $))$ 

```

Аннотируется?

Рис. 18: $append^o2$, полученное подстановкой $y \rightarrow x$ в $append^o$

2.6. 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 алгоритма анализа времени связывания для miniKanren. Для доказательства корректности необходимо показать его терминируемость и согласованность, что и сделано в последующих подчастях.

2.6.1. 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.

Тут учтены генерации и перестановки?

2.6.2. Согласованность

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

Проаннотировать тело отношения \leftrightarrow проаннотировать несколько дизъюнктов. Вычисление дизъюнктов в miniKanren происходит независимо, значит, и аннотировать их можно независимо. 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 $append^o$ 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 конз-

юнкта, 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.

3. Разработка транслятора

Эта глава посвящена разработке алгоритма трансляции miniKanren в абстрактный функциональный язык программирования. В первой части рассказывается об особенностях трансляции miniKanren. Абстрактное синтаксическое дерево функционального языка приводится во второй части. Третья часть описывает алгоритм трансляции. Ограничения подхода находятся в четвертой части.

Транслируете вы язык, не особенности

3.1. Особенности miniKanren и способы их трансляции

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 абстрактного 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 19.

```
8  appendo x1 y1 z0 =  
9  (x1 ≡ [] ∧  
10  y1 ≡ z0) ∨  
11  (fresh [h, t, r] (  
12    x3 ≡ h1 : t2 ∧  
13    z0 ≡ h1 : r1 ∧  
14    appendo t2 y2 r1  
15  ))
```

Рис. 19: 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 $R \subseteq X_0 \times \dots \times X_n$, с известными аргументами X_{i_0}, \dots, X_{i_k} , и неизвестными X_{j_0}, \dots, X_{j_l} , соответствует функция, возвращающая список результатов $F : X_{i_0} \rightarrow \dots \rightarrow X_{i_k} \rightarrow [X_{j_0} \times \dots \times X_{j_l}]$.

В качестве решения проблемы предлагается использовать кортежи. Пример трансляции *append^o* в обратном направлении приведён на рисунке 20. *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.

```

1      appendo00I x0 = appendo00IO x0 ++ appendo00I1 x0
2      appendo00IO s4@s0 = do
3          let s3 = []
4          return $ (s3, s0)
5      appendo00IO _ = []
6      appendo00I1 s4@(s5 : s7) = do
7          (s6, s0) <- appendo00I s7
8          let s3 = (s5 : s6)
9          return $ (s3, s0)
10     appendo00I1 _ = []

```

Рис. 20: Трансляция *append^o* в обратном направлении

3.1.2. Перекрытие 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, 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 20. Здесь *appendoOOI* является основной функцией, объединяющей результаты вспомогательных функций *appendoOOIO* и *appendoOOI1*. 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 *appendoOOIO*), а все последующие — из второго (функции *appendoOOI1*).

3.1.3. Недетерминированность результатов

miniKanren способен выдавать несколько вариантов одной переменной в качестве результата, создавая недетерминированность. Монада списка — способ поддержать недетерминированность в функциональном языке. Используем её в трансляции. Кро-

ме того, в Haskell для неё есть удобная нотация, называемая *do-нотацией*³.

Её применение так же можно видеть на примере трансляции *append*^o в обратном направлении, представленном на рисунке 20. 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. Порядок и направление при исполнении отношений

Самая главная особенность — порядок и направление исполнения отношений внутри целевого может отличаться для разных направлений. Данная проблема была указана во введении как мотивация к разработке алгоритма аннотирования переменных. Адаптация анализа времени связывания к *miniKanren* и есть решение данной проблемы. Разработке соответствующего алгоритма была посвящена вся предыдущая глава. В данном 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 19), которое необходимо транслировать. Результат трансляции есть на рисунке 20.

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 20) 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 12 обратится в определение *let* $x = (h : t)$, потому что аннотация x больше обеих аннотаций переменных h и t . Строка 13 даст определение *let* $(h : r) = z$. Направление вызова функции в строке 14 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: $(t, y) \leftarrow \text{append}^o \text{OOI } r$. Сортиров-

³Описание *do*-нотации языка Haskell: https://en.wikibooks.org/wiki/Haskell/do_notation, дата последнего посещения: 14.05.2020

ка определений позволит получить их следующий порядок: $(h : r)$, (t, y) , x . При трансляции они окажутся, соответственно, в 6, 7 и 8 строках.

3.1.5. 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.

Эта же проблема уже встречалась при реализации алгоритма аннотирования. В нём она разрешилась добавлением специальной унификации типа $x \equiv < gen :>$, где x — переменная, оставшаяся свободной, а $< gen :>$ — нотация, позволяющая транслятору понять необходимость генерации. Таким образом, всё, что остаётся сделать — правильно раскрыть нотацию и произвести генерацию.

Рассмотрим пример трансляции *append*^o в направлении с первой входной переменной (см. рисунок 21). 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 целых положительных чисел, а так же сам класс типов *Generator* представлены на рисунке 22.

```
22      appendoI00 x0 = appendoI000 x0 ++ appendoI001
      x0
23      appendoI000 s0@[ ] = do
24        s1 <- (gen)

25        let s2 = s1
26        return $ (s1, s2)
27      appendoI000 _ = []
28      appendoI001 s0@(s3 : s4) = do
29        (s1, s5) <- appendoI00 s4
30        let s2 = (s3 : s5)
31        return $ (s1, s2)
32      appendoI001 _ = []
```

Рис. 21: Трансляция *append*^o в направлении с первой входной переменной

3.1.6. Цикл только за счёт рекурсии

Данная особенность присуща и функциональной парадигме, поэтому проблем при трансляции с ней нет. Тем не менее, о ней стоило вспомнить для полноты обсуждения.

это происходит?

```

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..]

```

Этот генератор
никогда не
сгенерирует
список длины 2

Рис. 22: Класс типов *Generator* и его реализация для списков и целых чисел

3.2. Особенности трансляции

В предыдущем разделе были разобраны особенности трансляции, связанные с *miniKanren*. В этом — особенности, проявившиеся в процессе разработки алгоритма трансляции.

Все рассматриваемые в данном разделе примеры являются результатом трансляции отношения *append^o* в различных направлениях. Само отношение *append^o* на *miniKanren* представлено на рисунке 19.

3.2.1. Сопоставление с образцом для входных переменных

Сопоставление с образцом — хороший способ стиль "отсеять" заведомо ложные вычисления, используя информацию о типе конструктора аргумента. В качестве примера рассмотрим трансляцию *append^o* в прямом направлении (приведена на рисунке 23). Дизъюнкты исходного отношения *append^o* (см. рисунок ??) содержат унификации первого аргумента, являющегося входным: первый дизъюнкт — унификацию с пустым списком (строка 9), второй — с непустым (строка 12). 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 *appendIO0*.

Стоит отметить, зачем нужны 5 и 10 строки. Они представляют собой сопоставление с образцом, которое всегда завершится успехом, однако, они не влияют на результат вычисления, так как возвращают пустой список. *appendIO1*, 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 *appendIO1* завершится ошибкой за отсутствием возможности обработать соответствующий вход.

В случае, если в отношении на *miniKanren* одной переменной-аргументу соответствовало несколько унификаций, для неё появляется возможность выбрать, какая из

В начале была сквозная нумерация, тут уже нет. Один стиль выбрать и придерживаться

```

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. 23: Трансляция *append*^o в прямом направлении

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

3.2.2. Перекрытие 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 ?? есть унификации переменных *x* и *z*. Если мы будем транслировать *append*^o в направлении со входными первым и третьим аргументами, то получим перекрытие имён переменных. Переменная *h*, участвующая в обеих унификациях, перейдёт в оба сопоставления с образцом.

На рисунке 24 представлен результат трансляции *append*^o в обсуждаемом направлении. Второму дизъюнкту в нём соответствует функция *appendIOI1*, 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 одинаковую "голову". Восстановление этого условия происходит за счёт применения охранного выражения `| s3 == s2`.

первый элемент
списка

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.


```

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

```

Рис. 24: Трансляция *append^o* в направлении со входными первым и третьим аргументами

3.2.3. Перекрытие имён в определениях

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

На рисунке 25 представлено модифицированное отношение *append^o*. Оно связывает три списка таких, что первый является повтором первого элемента второго списка, а третий — конкатенацией первого и второго списков.

```

8     appendo Assign 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             appendo Assign t y r
17         ))

```

Рис. 25: Отношение *append^o Assign*

Рассмотрим результат его трансляции в обратном направлении, представленный на рисунке 26). 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 повторившуюся 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 функцию, необходимо знать, как

```

18     appendoAssign00I x0 = appendoAssign00I0 x0 ++ appendoAssign00I1 x0
19     appendoAssign00I0 s2@s1 = do
20         let s0 = []
21         return $ (s0, s1)
22     appendoAssign00I0 _ = []
23     appendoAssign00I1 s2@(s8 : (p2 : s9)) | s8 == p2 = do
24         let (s3 : s5) = s2
25         let (s6 : (c4 : s7)) = s2
26         (s4, s1) <- appendoAssign00I s5
27         let s0 = (s3 : s4)
28         if (s6 == c4) then return $ (s0, s1) else []
29     appendoAssign00I1 _ = []

```

Рис. 26: Трансляция *append^oAssign* в обратном направлении

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 27 приведен один из таких пользовательских конструкторов, реализующих натуральные числа.

```

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

```

Рис. 27: Тип данных *Peano* — определение конструкторов *O* и *S*

3.3. Алгоритм трансляции

Данный раздел посвящен самому алгоритму трансляции. Первая часть вводит абстрактный синтаксис в соответствии со всеми особенностями, затронутыми в двух предыдущих разделах. Вторая часть приводит общий алгоритм трансляции программы в абстрактном синтаксисе *miniKanren* в программу в абстрактном синтаксисе функционального языка.

3.3.1. Абстрактный синтаксис функционального языка

Транслированная программа *FuncProgram* представляет собой множество функций *F*.

```

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
44                  ] Expr
45
46      data F = F String [Line]
47
48      newtype FuncProgram = FuncProgram [F]

```

Рис. 28: Абстрактный синтаксис функционального языка

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 $Tuple$). Кортеж — 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 $Atom$.

3.3.2. Алгоритм трансляции

Первым шагом алгоритма трансляции, безусловно, будет запуск алгоритма аннотирования со всеми нормализациями. 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;
- Получение направлений конъюнктов; определение
- Получение порядка определений;
- Удаление перекрытий имён в определениях путём формирования условий для ветвления;

многие Множество из этих шагов ~~должны быть очевидны из рассмотренных~~ *были описаны* в предыдущих разделах особенностей miniKanren и трансляции. 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 значение 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. Так же как делали для унификаций, сохраним значение максимальной аннотации.

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

3.4. Корректность алгоритма

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. Однако, перебор все программ — алгоритмически неразрешимая задача.

Для доказательства корректности предлагается сделать следующее:

- Рассмотреть корректность перенесения особенностей miniKanren в функциональную парадигму;
- Проанализировать семантику конструкций miniKanren в терминах функциональной парадигмы;
- Проверить, что конструкции функционального языка, полученные из конструкций miniKanren, обладают той же семантикой;

Такой подход не гарантирует полностью корректный транслятор (что сделать невозможно), но позволит говорить о его корректности в первом приближении.

В разделе "Особенности miniKanren и способы их трансляции" рассмотрены способы транслировать особенности miniKanren в функциональную парадигму. Корректность этих способов вытекает из их определения.


Проанализируем, чем являются конструкции нормализованной программы на miniKanren для функциональной парадигмы:

- Fresh — объявление и определение переменной;
- Унификация или вызов отношения на полностью известных переменных — предикат, проверка соответствия переменных;
- Унификация — переопределение переменной: считаем, что при fresh переменная объявляется и получает своё значение — все возможные значения её типа;
- Вызов отношения — вызов функции, возвращает результат;
- Конъюнкт — переопределение или предикат;
- Дизъюнкт — функция, скоуп: внутри конъюнкции все конъюнкты, представляющие собой переопределения, могут использовать определённые ими переменные;
- Тело отношения — запуск нескольких функций и объединение их результатов;

Проанализируем результат трансляции для каждого из этих конструкций и проверим сохранение семантики:

- Унификация или вызов отношения на всех известных переменных — *if* или *guard*;
- Fresh + Унификация — сопоставление с образом или определение;
- Fresh + Вызов отношения — определение;
- Конъюнкт — определение, *if* или *guard*;
- Дизъюнкт — вспомогательная функция;
- Тело отношения — конкатенация результатов вызовов вспомогательных функций;

Таким образом мы показали, что для описанных конструкторов семантика сохраняется. Однако, мы не рассмотрели случай вызовов отношений, где все аргументы являются выходными. Такие программы запрещены для трансляции и на текущий момент являются ограничением подхода.



Я бы скорее в этом разделе поговорила о том, что означают направления, почему в двух крайних направлениях получаются генератор и предикат, а потом уже довершила тем, что наши конструкции совпадают по смыслу

4. Тестирование и анализ результатов

4.1. 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.

4.1.1. *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.

4.1.2. Классификация для аннотатора

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

- Нерекурсивные вызовы на аргументах-конструкторах;
- Рекурсивные вызовы на аргументах-конструкторах;
- Вызовы на одних и тех же переменных;

Все перечисленные виды поддерживаются.

4.1.3. Классификация по особенностям miniKanren

Как 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;
- Перекрытие в результатов дизъюнктов;
- Недетерминированность результатов;
- Унификация *fresh*-переменных только друг с другом;

Все перечисленные виды поддерживаются.

4.1.4. 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.

- Перекрытие имён в сопоставлениях с образцом (охранные выражения);
- Перекрытие 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.

4.2. Тестирование

В данный раздел посвящён описанию способа тестирования разработанного алгоритма трансляции.

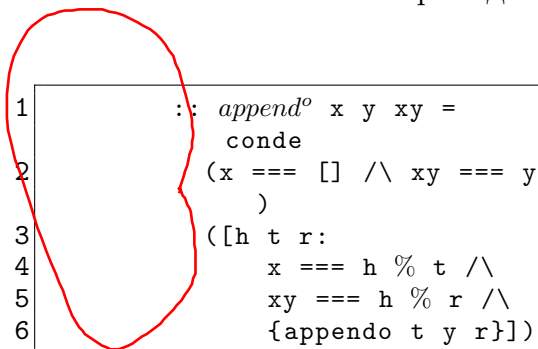
Он принимает на вход программу в абстрактном синтаксисе `miniKanren` и выдаёт программу в абстрактном синтаксисе функционального языка. Такой результат алгоритма невозможно запустить. В то же время наилучшим способом тестирования является именно запуск полученной функции. Это влечёт за собой необходимость 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.

4.2.1. 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`

4.2.2. 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.

4.2.3. Алгоритм тестирования

Для тестирования 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 функциональный синтаксис;
- Трансляция в Haskell;
- Запуск *unit*-тестов на транслированной Haskell-программе;

Результаты?
Вывод в духе
"работает, каньеш"

С целью
проверить, что
программа
обладает
желаемым
поведением.

Тут должна
быть ссылка на
репозиторий,
где тестовые
программы
лежат,
например

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 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.

- 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. Доказана корректность.

- Тестирование разработанного инструмента 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 и реализован его парсер, а так же транслятор абстрактного функционального синтаксиса в конкретный. Оба алгоритма реализованы на Haskell. Проанализированы причины невозможности трансляции некоторых программ.

Уточните,
принято ли так
писать в ИТМО.

Обычно цель и задачи не дублируются. Вводный абзац выглядит как-то в духе: "в рамках данной работы получены следующие результаты:". Дальше результаты в совершенной форме (типа "сделано", "разработано", "применено"). Подробное резюме ок, но с остальной обвязкой я бы уточнила.

Список литературы

мало

- [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] Thiemann Peter. A Unified Framework for Binding-Time Analysis // TAPSOFT. — 1997.
- [10] A Unified Approach to Solving Seven Programming Problems (Functional Pearl) / William E. Byrd, Usamichael Ballantyne, Usagregory Rosenblatt, Matthew Might // Relational Programming Workshop. — 2017.
- [11] Vanhoof Wim, Bruynooghe Maurice, Leuschel Michael. Binding-time analysis for Mercury // Program Development in Computational Logic. — Springer, 2004. — P. 189–232.

А. Грамматика языка miniKanren

```
 $\langle program \rangle ::= \langle def \rangle^* \langle goal \rangle$   
 $\langle term \rangle ::= \langle ident \rangle \mid \text{'<'} \langle ident \rangle \text{' : ' } \langle term \rangle^* \text{'>'}$   
 $\langle def \rangle ::= \text{'::'} \langle ident \rangle \langle ident \rangle^* \text{'=' } \langle goal \rangle$   
 $\langle goal \rangle ::= \langle disj \rangle \mid \langle fresh \rangle \mid \langle invoke \rangle$   
 $\langle fresh \rangle ::= \text{'[' } \langle ident \rangle^+ \text{' : ' } \langle goal \rangle \text{' ]'}$   
 $\langle invoke \rangle ::= \text{'{' } \langle ident \rangle \langle term \rangle^* \text{'}'}$   
 $\langle disj \rangle ::= \langle conj \rangle (\text{'\/' } \langle conj \rangle)^*$   
 $\langle conj \rangle ::= \langle pat \rangle (\text{'/\'} \langle pat \rangle)^*$   
 $\langle pat \rangle ::= \langle term \rangle \text{'===' } \langle term \rangle \mid \langle fin \rangle$   
 $\langle fin \rangle ::= \langle fresh \rangle \mid \text{'(' } \langle disj \rangle \text{' ')'}$ 
```