

# Анализ времени связывания для реляционных программ

Ирина Артемьева  
Университет ИТМО  
Санкт-Петербург, Россия  
irinapluralia@gmail.com

Екатерина Вербицкая  
JetBrains Research  
Санкт-Петербург, Россия  
kajigor@gmail.com

**Аннотация**—Программы в парадигме реляционного программирования представляют собой математические отношения. Программы-отношения можно исполнять в различных направлениях: зафиксировав часть аргументов программы, находить значение остальных. Не всегда исполнение программы в заданном направлении эффективно. Одним из способов улучшения производительности является трансляция реляционных программ в функциональные. Для генерации функции по отношению необходимо определить порядок связывания имен в программе с учетом заданного направления. Для этого традиционно применяется анализ времени связывания, однако для реляционных языков ранее его разработано не было. В статье мы предлагаем алгоритм анализа времени связывания для языка реляционного программирования `miniKANREN`.

**Ключевые понятия**—Реляционное программирование, анализ времени связывания, статический анализ

## I. ВВЕДЕНИЕ

Реляционное программирование — парадигма, в которой любая программа описывает математическое отношение на ее 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 входной аргумент будем обозначать  $?$ ). Выполнение отношения  $add^o x y ?$  с зафиксированными (выходными) 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 некоторое конкретное направление, называемое прямым (например,  $add^o x y ?$ ), все остальные направления обычно называются обратными. Возможность выполнения в различных направлениях — основное преимущество реляционного програм-

мирования. 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<sup>1</sup> [1]. Основным представителем парадигмы реляционного программирования является семейство интерпретируемых языков `miniKanren`<sup>2</sup>. Языки семейства `miniKanren` компактны и встраиваются в языки общего назначения, за счет чего их проще использовать в своих проектах. Для встраивания достаточно реализовать интерпретатор языка `miniKanren`: ядро языка на Scheme занимает не более, чем 40 строк [2]. Помимо этого, `miniKanren` реализует полный поиск со стратегией interleaving, поэтому любая программа, написанная на нем, найдет все существующие ответы, в то время как Prolog может никогда не завершить поиск. В этой статье мы будем говорить про `miniKanren`.

Возможность выполнения программ на `miniKanren` в различных направлениях позволяет решать задачи поиска посредством решения задачи распознавания [3]. Так, имея интерпретатор языка, можно решать задачу синтеза программ на этом языке по набору тестов [4]; 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 [3] (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 [3] для решения этой проблемы используется специализация.

<sup>1</sup>Официальный сайт языка MERCURY: <https://mercurylang.org/>, дата последнего посещения: 15.02.2020

<sup>2</sup>Официальный сайт языка `miniKANREN`: <http://minikanren.org/>, дата последнего посещения: 15.02.2020

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 [5]. К сожалению, реализация такого специализатора — нетривиальная задача.

В данное время авторами ведется работа над альтернативным подходом улучшения производительности программы в заданном направлении. Для этого по отношению с заданным направлением генерируется функция на функциональном языке программирования Haskell. Таким образом можно избежать затрат на интерпретацию. Особенностью реляционного программирования является отсутствие строго порядка исполнения программы: особенно сильно он может различаться для разных направлений. Это затрудняет трансляцию в функциональные языки программирования. Для успешной трансляции необходимо определить порядок исполнения программ с учетом направления. Для решения такой задачи используется анализ времени связывания (binding time analysis). Функционально-логический язык программирования Mercury использует анализ времени связывания как шаг компиляции [6], однако для реляционных языков ранее не применялся. В данной статье мы представляем алгоритм времени связывания для реляционного программирования.

В разделе II мы описываем язык miniKanren, используемый в статье. Раздел III содержит схему его трансляции в функциональный язык, а также описание возникающих при этом трудностей. Алгоритм анализа времени связывания для miniKanren приведен в разделе IV. В заключении (раздел V) мы подводим выводы и описываем планы на дальнейшую работу.

## II. ЯЗЫК ПРОГРАММИРОВАНИЯ miniKANREN

Семейство языков miniKanren дало рождение парадигме реляционного программирования. Это минималистичные языки, встраиваемые в языки программирования общего назначения. 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. 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.

```
Goal : Goal ∨ Goal
      | Goal ∧ Goal
      | Term ≡ Term
      | invoke Name [Term]
      | fresh [Var] Goal
Term : Var
      | cons Name [Term]
```

Пример программы на языке miniKanren, связывающей три списка, где третий является конкатенацией первых двух, приведен ниже. Мы используем [] как сокращение для пустого списка (cons Nil []) и  $h : t$  для конструктора списка с головой  $h$  и хвостом  $t$  (cons Cons  $[h, t]$ ), а  $[x_0, x_1, \dots, x_n]$  — для обозначения списка с элементами  $x_0, x_1, \dots, x_n$ .

```
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 `appendo [1,2] [3] ?` вернет 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 `appendo ? ? [1,2,3]` является множество пар  $\{([], [1, 2, 3]), ([1], [2, 3]), ([1, 2], [3]), ([1, 2, 3], [])\}$ .

## III. ТРАНСЛЯЦИЯ В ФУНКЦИОНАЛЬНЫЙ ЯЗЫК

В этом разделе мы кратко опишем разрабатываемую авторами трансляцию miniKanren в функциональный язык программирования, чтобы продемонстрировать, на решение каких проблем нацелен анализ времени связывания. Мы будем использовать Haskell в качестве целевого языка.

Отношение, выполненное в заданном направлении, можно рассматривать как функцию из известных аргументов в неизвестные. Например, отношение `appendo`,

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.

```

1   $append^o :: [a] \rightarrow [a] \rightarrow [a]$ 
2   $append^o []\ y = y$ 
3   $append^o (h : t)\ y =$ 
4    let  $r = append^o\ t\ y$  in
5     $h : r$ 
```

Рис. 2. Результат трансляции  $append^o\ x\ y\ ?$

Не всегда результатом выполнения отношения является единственный ответ. Например, при выполнении отношения  $append^o$  в обратном направлении ( $append^o\ ?\ ?\ z$ ), miniKanren вычислит все возможные пары списков, дающие при конкатенации  $z$ . В общем случае, отношению  $R \subseteq X_0 \times \dots \times X_n$ , в котором известны аргументы  $X_{i_0}, \dots, X_{i_k}$ , а аргументы  $X_{j_0}, \dots, X_{j_l}$  необходимо вычислить, соответствует функция  $F : X_{i_0} \rightarrow \dots \rightarrow X_{i_k} \rightarrow [X_{j_0} \times \dots \times X_{j_l}]$ , возвращающая список результатов.

Любое отношение можно преобразовать в нормальную форму. Для упрощения повествования мы будем считать, что все цели нормализованы. Нормальной формой будем называть дизъюнкцию конъюнкций вызовов отношений или унификаций термов, при этом все свободные переменные введены в область видимости в самом начале:

$$\begin{aligned}
& Goal : \underline{fresh}\ [Name] (\bigvee \bigwedge Goal') \\
& Goal' : \underline{invoke}\ Name\ [Term] \\
& \quad | Term \equiv Term
\end{aligned}$$

Транслятор строит одну функцию для каждого дизъюнкта. Дизъюнкты в программе на miniKanren независимы, то есть все ответы из каждого дизъюнкта объединяются для получения результата выполнения

отношения. Для отношения создается функция, конкатенирующая результаты применения функций, построенных для отдельных дизъюнктов.

Пример трансляции  $append^o\ ?\ ?\ z$  приведен на листинге 3. Унификации неизвестных переменных (например  $x \equiv []$  и  $x \equiv (h : t)$ ) при трансляции преобразуются в let-связывания (строки 5 и 11). Рекурсивные вызовы отношений транслируются в рекурсивные вызовы функций, построенных в заданном направлении (см. строку 10). Стоит обратить внимание на то, что рекурсивно вызывается функция  $append^o$ , построенная по всему отношению. Мы используем do-нотацию языка Haskell<sup>3</sup>. Связывание в строке 10 означает, что результат будет вычислен для каждого элемента списка, полученного из рекурсивного вызова функции.

```

1   $append^o :: [a] \rightarrow [([a], [a])]$ 
2   $append^o\ x = append_1^o\ x ++ append_2^o\ x$ 
3  where
4     $append_1^o\ y = \mathbf{do}$ 
5      let  $x = []$ 
6      return  $(x, y)$ 
7     $append_1^o\ _ = []$ 
8
9     $append_2^o\ (h : r) = \mathbf{do}$ 
10      $(t, y) \leftarrow append^o\ r$ 
11     let  $x = h : t$ 
12     return  $(x, y)$ 
13    $append_2^o\ _ = []$ 
```

Рис. 3. Результат трансляции  $append^o\ ?\ ?\ z$

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. 1, строка 6), в то время как в функциях выполняется в первую очередь. Если отношение вызывает более одного отношения, то необходимо не только определить порядок, в котором необходимо вызывать функции в результате трансляции, но и в каком направлении это делать. Примером может служить отношение  $revers^o$  (см. листинг 4), связывающий список со списком его элементов в обратном порядке. Это отношение имеет рекурсивный вызов, а также вызов отношения  $append^o$ . Порядок вызовов здесь влияет на направления функций, построенных по этим отношениям. Выбранные направления также могут влиять на то, в каком порядке необходимо вызывать функции. Использование монадических вычислений и do-нотации вынуждает нас заранее определять порядок, в котором будут осуществляться вызовы функций.

<sup>3</sup>Описание do-нотации языка Haskell: [https://en.wikibooks.org/wiki/Haskell/do\\_notation](https://en.wikibooks.org/wiki/Haskell/do_notation), дата последнего посещения: 14.02.2020

```

1  reverso xs sx =
2    (xs ≡ [] ∧ sx ≡ []) ∨
3    (fresh [h, ts, st] (
4      x ≡ h : ts ∧
5      reverso ts st ∧
6      appendo [st, [h], sx]
7    ))

```

Рис. 4. Отношение  $revers^o$

Эти особенности диктуют необходимость использования некоторого статического анализа, позволяющего упорядочить вычисления в заданном направлении. Анализ времени связывания часто используется при построении offline-специализаторов языков программирования. Его задачей является определить, являются ли данные, используемые в программах, статическими (известными заранее) или динамическими (известными только во время вычисления). Использование анализа времени связывания при функциональной трансляции может также определить порядок вычисления, а по нему — направлениях, в которых необходимо транслировать используемые отношения.

#### IV. 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.

Если о переменной ничего неизвестно — она аннотируется *Undef*; иначе указывается время связывания — целое положительное число. В начале работы алгоритма известными являются переменными, указанные как входные — они аннотируются числом 0. Если переменная унифицируется с константой (термом, не содержащим свободных переменных), то мы считаем временем ее 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 новую аннотацию, если появилась какая-то новая информация о времени ее связывания. При этом аннотация никогда не заменяется на меньшую.

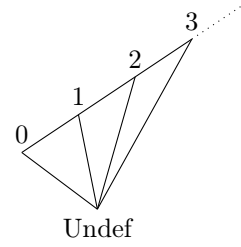


Рис. 5. Полурешётка на аннотациях

#### А. Алгоритм анализа времени связывания

Реализация разработанного алгоритма доступна на сайте GitHub<sup>4</sup>. Ниже приведено описание алгоритма.

Входные данные алгоритма: программа на miniKanren (цель) и список входных переменных. Выходные данные — пара из проаннотированной нормализованной (приведённой к дизъюнктивной нормальной форме) цели и списка проаннотированных определений отношений, вызванных цели. Мы будем называть этот список стеком вызова, потому что в нем будут находиться вызываемые отношения.

При инициализации алгоритма выполняются следующие действия:

- все *fresh*-переменные уникально переименовываются, чтобы избежать перекрытия имён;
- цель приводится в дизъюнктивную нормальную форму;
- все входные переменные аннотируются 0;
- создается пустой стек вызовов.

Каждый раз при обработке вызова отношения анализируется его тело, при этом:

- *fresh*-переменные уникально переименовываются;
- цель приводится в нормальную форму;
- производится первичное аннотирование цели данными о входных переменных.

Аннотация цели осуществляется итеративно, пока не будет достигнута неподвижная точка функции, описывающей шаг аннотирования. За один шаг аннотирования мы аннотируем либо одну унификацию, либо один вызов отношения. Для аннотации цели в нормальной форме необходимо проаннотировать все ее 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.

<sup>4</sup>Исходный код алгоритма аннотирования: [https://github.com/Pluralia/uKanren\\_translator](https://github.com/Pluralia/uKanren_translator), дата последнего посещения: 15.02.2020

При аннотировании унификаций возможны следующие случаи.

- 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 проаннотированы  $Undef$ . Тогда всем переменным  $y_j^{Undef}$  присваивается аннотация  $n + 1$ .
- Оба унифицируемых терма — конструкторы:  $\underline{cons} \text{ Name } [t_0^{i_0}, \dots, t_k^{i_k}] \equiv \underline{cons} \text{ Name } [s_0^{j_0}, \dots, s_k^{j_k}]$ . В этом случае имена конструкторов должны быть одинаковыми, а количество термов совпадать — иначе такие термы унифицировать не удастся. Такая унификация эквивалентна конъюнкции унификаций вида  $t_l^{i_l} \equiv s_l^{j_l}$ , каждую из которых следует анализировать в соответствии с одним из перечисленных случаев.
- Остальные случаи симметричны.

Помимо унификации конъюнкт может быть вызовом некоторого отношения на частично проаннотированных термах. Для аннотации вызова мы рассматриваем тело соответствующего отношения и аннотируем его. Для избежания повторного аннотирования информация о ранее проаннотированных в конкретных направлениях отношениях сохраняется в стеке вызовов. Если частично определенное направление текущего вызова согласовано с ранее проаннотированным, его аннотировать не нужно. Два направления назовём согласованными, если при сравнении аннотаций их аргументов попарно аннотации одного направления будут всегда не меньше соответствующих аннотаций другого.

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 согласованными, так как аннотации переменной  $x$  упорядочены правильно ( $0 < 1$ ), а для  $y$  и  $z$  аннотации совпадают. В то же время, направление  $r^o x^3 y^2 z^0$  не согласовано с направлением  $r^o x^0 y^0 z^2$ , так как аннотации  $x$  и  $y$  больше у первого направления, а аннотация  $z$  больше у второго.

При аннотации вызовов отношений возможные следующие случаи.

- Переменные всех аргументов проаннотированы  $Undef$ . В этом случае для аннотирования не достаточно информации, поэтому следует перейти к аннотации следующего конъюнкта.
- Вызов полностью проаннотирован, то есть аннотация всех переменных — числа. Здесь дальнейшая аннотация не требуется.

- Вызов с таким именем и согласованным направлением уже есть в стеке вызовов — заменить  $Undef$ -аннотации переменных на  $n + 1$ , где  $n$  — максимальная аннотация переменных согласованного направления.
- Вызова с таким именем и направлением нет в стеке вызовов. В этом случае мы сначала добавляем его и направление в стек вызовов. Затем аннотируем тело вызываемого отношения с учётом обновлённого стека. По завершении аннотации добавить в стек проаннотированную цель.

Существуют отношения, точная аннотация которых не возможна без вмешательства человека или полного перебора возникающих вариантов. В таких отношениях некоторые переменные используются только в отношениях, не участвуя в унификациях. Пример такого отношения приведён в листинге 6. 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 или после. При вычислении между ними получим вычисление  $g^o$  в прямом или обратном порядке, при вычислении после них  $g^o$  выполняет роль предиката.

1	$rel^o \ x \ y \ z =$
2	$f^o \ x \ y \ \vee$
3	$h^o \ z \ y \ \vee$
4	$g^o \ x \ z$

Рис. 6. Пример программы на miniKANREN, в которой переменные используются только в отношениях

При получении такого отношения алгоритм аннотирования возвращает частично проаннотированную цель (некоторые переменные будут иметь аннотацию  $Undef$ ). В этом случае мы запускаем алгоритм еще раз, изменив порядок вызова отношений в соответствующем дизъюнкте. Если и при другом порядке в аннотированной цели останутся  $Undef$ -переменные, цель считается неаннотируемой.

Предложенный алгоритм терминируется, так как повторное аннотирование отношений не производится (имеющиеся в стеке вызовов отношения не аннотируются снова), при этом в каждом отношении используется конечное количество уникальных переменных. Это значит, что каждому отношению можно сопоставить конечное количество уникальных аннотаций.

### В. Примеры аннотирования

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

1) *Отношение  $append^o$  в прямом направлении:*

В данном случае переменные  $x$  и  $y$  являются входными. При начале работы алгоритма, его нет в стеке вызовов, поэтому добавим его и запустим рекурсивно аннотирование цели  $append^o$ . Проаннотированное  $append^o$  приведено в листинге 7. 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 имеющимся в стеке, поэтому можно проаннотировать переменную  $r$ . Распространяем аннотацию  $r$  в строке 5. На последнем шаге аннотируем  $z$  в строке 4.

После аннотации обоих дизъюнктов остается определить аннотация для исходного отношения. Для этого каждому аргументу мы присваиваем аннотацию, равную максимальной его аннотации среди всех дизъюнктов. В первом дизъюнкте переменная  $z$  имеет аннотацию 1, а во втором — 3, поэтому результирующая аннотация равна 3.

```

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

```

Рис. 7. Аннотирование  $append^o$  в прямом направлении

2) *Отношение  $append^o$  в обратном направлении:*

В этом случае мы считаем переменную  $z$  входной (см. листинг 8). 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 5). Аннотация  $r$  распространяется на строку 6. Отношение с согласованным направлением есть в стеке, поэтому можно аннотировать  $t$  и  $y$ . Далее аннотация  $t$  распространяется на строку 4, и на последнем шаге аннотируется  $x$ .

3) *Отношение  $revers^o$  в обратном направлении:*

Отношение  $revers^o$  связывает два списка, получающиеся переворачиванием друг друга. Его определение приведено в листинге 9.

Добавим  $revers^o$  по обратному направлению в стек вызовов, проинициализируем  $y$  — как входную переменную. Рассмотрим только второй дизъюнкт. На первом шаге можно попытаться проаннотировать только вызов  $append^o$  в строке 6 — известна  $y$ . Такого отношения в стеке вызовов нет — добавляем и вызываем аннотирование. Это и есть вызов  $append^o$  в обратном

```

1  appendo x3 y2 z0 =
2  (x1 ≡ [] ∧ y1 ≡ z0) ∨
3  (fresh [h, t, r] (
4    x3 ≡ h1 : t2 ∧
5    z0 ≡ h1 : r1 ∧
6    appendo [t2, y2, r1]
7  ))

```

Рис. 8. Аннотирование  $append^o$  в обратном направлении

направлении, рассмотренный выше. 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 5). Распространяем аннотацию  $t$  и аннотируем  $x$  на следующем шаге в строке 4.

```

1  reverso x5 y0 =
2  (x1 ≡ [] ∧ y1 ≡ []) ∨
3  (fresh [h, t, r] (
4    x5 ≡ h2 : t4 ∧
5    reverso [t4, r3] ∧
6    appendo [r3, [h2], y0]
7  ))

```

Рис. 9. Аннотирование  $revers^o$  в обратном направлении

### С. Обзор существующих решений

Анализ времени связывания часто используется при offline-специализации программ [5]. 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. В зависимости от цели применения могут выбираться разные домены времен связывания. Анализ времени связывания существует для логического языка Prolog и функционального-логического языка Mercury — представителей родственных реляционному программированию парадигм.

В языке Mercury анализ времени связывания используется для эффективной компиляции. При этом определяется только

Так, в Mercury [6] анализ времени связывания применяется для специализации программ. Как следствие, этой реализации не требуется размечать порядок связывания переменных — достаточно разбить их на статические и динамические. Особенностью Mercury являются типы, что так же отразилось на итоговом алгоритме (аннотирование происходит с учётом графа типов, построенного по термам).

Существует анализ времени связывания для лямбда-исчисления [7]. Его цель совпадает с нашей — получить

порядок связывания переменных. В алгоритме, предложенном в статье, для этого используются числа от 0 до  $D$  — эту идею получилось перенести в анализ времени связывания для miniKanren. Однако, лямбда-исчисление — функциональное программирование — использует совершенно другие конструкции.

## V. ЗАКЛЮЧЕНИЕ

В статье мы представили алгоритм анализа времени связывания для miniKanren. Основной его недостаток — полный перебор при аннотации переменных, если они используются только в вызовах отношений, и не были проаннотированы ранее. В этом случае необходимо перебрать все возможные направления вычисления отношений, что влияет на эффективность алгоритма.

Также в дальнейшем мы планируем интегрировать анализ времени связывания в транслятор в функциональный язык. По проаннотированной программе можно получить порядок, в котором необходимо привести определения переменных и вызовы функций.

## БЛАГОДАРНОСТЬ

Выражаем благодарность Дмитрию Юрьевичу Булычеву и Даниилу Андреевичу Березуну за плодотворные дискуссии и конструктивную критику.

## СПИСОК ЛИТЕРАТУРЫ

- [1] Z. Somogyi, F. Henderson, and T. Conway, “The execution algorithm of mercury, an efficient purely declarative logic programming language,” *The Journal of Logic Programming*, vol. 29, no. 1, pp. 17 – 64, 1996, high-Performance Implementations of Logic Programming Systems. [Online]. Available: <http://www.sciencedirect.com/science/article/pii/S0743106696000684>
- [2] J. Hemann and D. P. Friedman, “ukanren: A minimal functional core for relational programming,” 2013.
- [3] P. Lozov, E. Verbitskaia, and D. Boulytchev, “Relational interpreters for search problems,” in *Relational Programming Workshop*, 2019, p. 43.
- [4] W. E. Byrd, U. Ballantyne, U. Rosenblatt, and M. Might, “A unified approach to solving seven programming problems (functional pearl),” in *Relational Programming Workshop*, 2017.
- [5] N. D. Jones, C. K. Gomard, and P. Sestoft, *Partial evaluation and automatic program generation*. Peter Sestoft, 1993.
- [6] W. Vanhoof, M. Bruynooghe, and M. Leuschel, “Binding-time analysis for mercury,” in *Program Development in Computational Logic*. Springer, 2004, pp. 189–232.
- [7] P. Thiemann, “A unified framework for binding-time analysis,” in *TAPSOFT*, 1997.