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 учётом 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 её аргументах. Имея программу-отношение, можно выполнять запросы: указывая некоторые известные аргументы, получать значения остальных. Например, $add^o \subseteq Int \times Int \times Int$ описывает отношение, третий аргумент которого является суммой первых двух. Рассмотрим возможные направления вычисления этого отношения (здесь и далее искомый аргумент будем обозначать знаком "?"). 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 Mercury[1] [1]. Основным представителем парадигмы реляционного программирования является семейство интерпретируемых языков miniKanren[2]. Языки семейства miniKanren компактны и встраиваются в языки общего назначения, за счёт чего их 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 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.

[1] Официальный сайт языка Mercury: https://mercurylang.org/, дата последнего посещения: 15.02.2020

[2] Официальный сайт языка miniKanren: http://minikanren.org/, дата последнего посещения: 15.02.2020

Copyright © 2020 for this paper by its authors.
Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

CEUR-WS.org/Vol-2691/paper51.pdf

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 фиксированным 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 отличаться для разных направлений. Это затрудняет трансляцию в функциональные языки программирования. Для успешной трансляции необходимо определить порядок исполнения программ с учётом 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 такой анализ 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 описывает схему его трансляции в функциональный язык и возникающие при этом трудности. Алгоритм анализа времени связывания для MINIKANREN приведен в разделе IV. Обзор литературы — в разделе V. В заключении (раздел VI) подведены итоги и описаны 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 и цели. Определение имеет имя, список аргументов и тело. Тело отношения является *целью*, которая может содержать *унификацию термов* и *вызовы отношений*, скомбинированные при помощи *дизъюнкций* и *конъюнкций*. *Терм* представляет собой или *переменную*, или *конструктор* с именем и списком подтермов. Свободные переменные вводятся в область видимости при помощи конструкции $\underline{fresh}$. Абстрактный синтаксис языка приведен ниже:

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

Пример программы на языке MINIKANREN, связывающей три списка, где третий является конкатенацией первых двух, приведен на рис. 1. Для краткости [] заменяет пустой список ($\underline{cons}\ Nil$ []); $h : t$ обозначает список с головой h и хвостом t ($\underline{cons}\ Cons\ [h, t]$), а $[x_0, x_1, \ldots, x_n]$ — список с элементами $x_0, x_1, \ldots, x_n$. Вызов отношения $\underline{call}\ relation\ [t_0, \ldots t_k]$ записывается как $relation\ t_0 \ldots\ t_k$.

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

Рис. 1. Пример программы на MINIKANREN

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

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 $append^o$, 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 рис. 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  appendᵒ :: [a] → [a] → [a]
 9  appendᵒ [] y = y
10  appendᵒ (h : t) y =
11    let r = appendᵒ t y in
12    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 $R \subseteq X_0 \times \cdots \times X_n$, с известными аргументами $X_{i_0}, \ldots X_{i_k}$, и неизвестными $X_{j_0}, \ldots X_{j_l}$, соответствует функция, возвращающая список результатов $F : X_{i_0} \to \cdots \to X_{i_k} \to [X_{j_0} \times \cdots \times X_{j_l}]$.

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, в которой все свободные переменные введены в область видимости в самом начале:

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

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 рис. 3. Унификации неизвестных переменных (например $x \equiv []$ и $x \equiv h : t$) при трансляции преобразуются в let-связывания (строки 17 и 23). Рекурсивные вызовы отношений транслируются в рекурсивные вызовы функций, построенных в заданном направлении (см. строку 22). Стоит обратить внимание, что рекурсивно вызывается функция $append^o$, построенная по всему отношению. Мы используем do-нотацию языка Haskell[3]. Связывание в строке 22 означает, что результат будет вычислен для каждого элемента списка, полученного из рекурсивного вызова функции.

```
13  appendᵒ :: [a] → [([a], [a])]
14  appendᵒ x = appendᵒ₁ x ++ appendᵒ₂ x
15    where
16      appendᵒ₁ y = do
17        let x = []
18        return (x, y)
19      appendᵒ₁ _ = []
20
21      appendᵒ₂ (h : r) = do
22        (t, y) ← appendᵒ r
23        let x = h : t
24        return (x, y)
25      appendᵒ₂ _ = []
```

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. рис. 4), связывающее 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.

Эти особенности диктуют необходимость использования некоторого статического анализа, позволяющего упорядочить вычисления в заданном направлении.

[3] Описание do-нотации языка Haskell: https://en.wikibooks.org/wiki/Haskell/do_notation, дата последнего посещения: 15.02.2020

```
26  reversᵒ xs sx =
27     (xs ≡ [] ∧ sx ≡ []) ∨
28     (fresh [h, ts, st] (
29          x ≡ h : ts ∧
30          reversᵒ ts st ∧
31          appendᵒ st [h] sx
32     ))
```

Рис. 4. Отношение $revers^o$

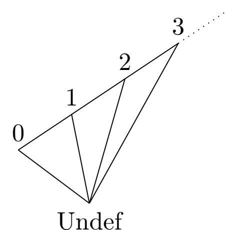

Рис. 5. Полурешетка на аннотациях

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 — направления, 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 переменные, указанные как входные — они аннотируются числом 0. Если переменная унифицируется с константой (термом, не содержащим свободных переменных), то мы считаем 1 её временем связывания. 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.

A. Алгоритм анализа времени связывания

Реализация разработанного алгоритма доступна на сайте GitHub[4]. Ниже приведено описание алгоритма.

Входные данные алгоритма: программа на miniKanren (цель) и список входных переменных. Выходные данные — пара из проаннотированной нормализованной (приведенной к дизъюнктивной нормальной форме) цели и списка проаннотированных определений отношений, вызываемых целью. 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 имен;
- 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 мы аннотируем либо одну унификацию, либо один вызов отношения. Для аннотации цели в нормальной форме необходимо проаннотировать все её дизъюнкты. Аннотации переменных в дизъюнкте должны согласовываться: одна и та же переменная в конъюнктах одного дизъюнкта должна иметь одну и ту же аннотацию. Конъюнкты аннотируются в заранее определенном порядке. Сначала мы аннотируем унификации, а затем вызовы отношений. Каждый раз при аннотации новой переменной необходимо установить ту же аннотацию всем другим вхождениям этой переменной в дизъюнкте.

[4]Исходный код алгоритма аннотирования: https://github.com/Pluralia/uKanren_translator, дата последнего посещения: 15.02.2020

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

- Унификация имеет вид $x^{Undef} \equiv t[y_0^{i_0}, \ldots, y_k^{i_k}]$, то есть переменная, имеющая аннотацию $Undef$, унифицируется с термом t со свободными переменными $y_j^{i_j}$ с целочисленными аннотациями i_j. В таком случае переменной x необходимо присвоить аннотацию $n+1$, где $n = max\{i_0, \ldots i_k\}$.
- Переменная, аннотированная числом, унифицируется с термом: $x^n \equiv t[y_0^{i_0}, \ldots, y_k^{i_k}]$; некоторые свободные переменные терма проаннотированы $Undef$. Тогда всем переменным y_j^{Undef} присваивается аннотация $n+1$.
- Оба унифицируемых терма — конструкторы: $\underline{cons}\ Name\ [t_0^{i_0}, \ldots, t_k^{i_k}] \equiv \underline{cons}\ Name\ [s_0^{j_0}, \ldots, s_k^{j_k}]$. 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, анализировать его не нужно. Два направления назовем *согласованными*, если при попарном сравнении аннотаций их аргументов 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. При этом, 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 в первом направлении, а z — во втором.

При аннотации вызовов отношений возможны 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 обновленного стека. По завершении аннотации добавляем 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 приведен на рис. 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 (в прямом или обратном порядке) или после (выполнять роль предиката). Стоит отметить, что на практике такие ситуации возникают достаточно редко.

```
33   rel^o x y z =
34     f^o x y ∧
35     h^o z y ∧
36     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, а в каждом отношении используется конечное количество уникальных переменных. Это значит, что каждому отношению можно сопоставить конечное количество уникальных аннотаций.

B. Примеры аннотирования

В этом разделе приведено 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, таких отношения и направления нет в стеке вызовов, поэтому добавим их и запустим рекурсивно аннотирование цели $append^o$. Проаннотированное $append^o$ приведено на рис. 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 40 можно установить, так как известна аннотация x. Аннотация h распространяется на 41 строку, а аннотация t — на 42 строку. Рекурсивный вызов отношения в строке 42 согласован с имеющимся в стеке, поэтому можно проаннотировать переменную r. Распространяем аннотацию r в строке 41. На последнем шаге аннотируем z в строке 40.

После аннотации обоих дизъюнктов остается определить аннотации 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.

```
37  append^o x^0 y^0 z^3 =
38    (x^0 ≡ [] ∧ y^0 ≡ z^1) ∨
39    (fresh [h, t, r] (
40        x^0 ≡ h^1 : t^1 ∧
41        z^3 ≡ h^1 : r^2 ∧
42        append^o t^1 y^0 r^2
43    ))
```

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

2) Отношение $append^o$ в обратном направлении: В этом случае мы считаем переменную z входной (см. рис. 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 48). Аннотация r распространяется на строку 49. Отношение с согласованным направлением есть в стеке, поэтому можно аннотировать t и y. Далее аннотация t распространяется на строку 47, 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 на рис. 9.

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 56 — известна y. Такого отношения в

```
44  append^o x^3 y^2 z^0 =
45    (x^1 ≡ [] ∧ y^1 ≡ z^0) ∨
46    (fresh [h, t, r] (
47        x^3 ≡ h^1 : t^2 ∧
48        z^0 ≡ h^1 : r^1 ∧
49        append^o t^2 y^2 r^1
50    ))
```

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

стеке вызовов нет — добавляем и вызываем аннотирование. Это и есть вызов $append^o$ в обратном направлении, рассмотренный выше (см. рис. 8). 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 55). Распространяем аннотацию t и аннотируем x на следующем шаге в строке 54.

```
51  revers^o x^5 y^0 =
52    (x^1 ≡ [] ∧ y^1 ≡ []) ∨
53    (fresh [h, t, r] (
54        x^5 ≡ h^2 : t^4 ∧
55        revers^o t^4 r^3 ∧
56        append^o r^3 [h^2] y^0
57    ))
```

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

V. 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 [7] и функционального-логического языка Mercury [6] — представителей родственных реляционному программированию парадигм.

В языке Mercury анализ времени связывания [6] 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 [7]. Основное предназначение анализа в этой работе — улучшение качества специализации, это решение нельзя использовать для определения правильного порядка вызовов отношений.

Работа [8] описывает анализ времени связывания для лямбда-исчисления с функциями высшего порядка. Его цель также в том, чтобы определить порядок связывания переменных, поэтому авторы используют отрезок натурального ряда $\{0, 1, \ldots, N\}$. Мы использовали эту идею в нашей работе. Помимо этого работа [8] описывает локальный подход для анализа времени связывания, который дает более точные результаты. Адаптация этого подхода для miniKanren — предмет дальнейшего исследования.

VI. Заключение

В статье мы представили алгоритм анализа времени связывания для miniKanren. Он определяет порядок, в котором связываются переменные данного отношения с учётом направления его вычисления.

Основной его недостаток — полный перебор при аннотации не аннотированных ранее переменных в случае их использования только в вызовах отношений. В этом случае необходимо перебрать все возможные направления вычисления отношений, что влияет на эффективность алгоритма.

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

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.

[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] M. Leuschel, S.-J. Craig, M. Bruynooghe, and W. Vanhoof, “Specialising interpreters using offline partial deduction,” vol. 3049, 01 2004, pp. 340–375.

[8] P. Thiemann, “A unified framework for binding-time analysis,” in *TAPSOFT*, 1997.

Binding-Time Analysis for Relational Programs

Irina Artemeva
ITMO University
Saint Petersburg, Russia
irinapluralia@gmail.com

Ekaterina Verbitskaia
JetBrains Research
Saint Petersburg, Russia
kajigor@gmail.com

Abstract—Programs in relational programming are mathematical relations. Such relations can be run in different directions: by providing some arguments of a program, one can compute the values of the others. The execution of a program in the given direction is not always efficient. One way to improve the performance of a relational program is to convert it into a functional program. To create a function by a relation, it is necessary to determine the order in which names within the input program are bound with respect to the given direction. Binding-time analysis is used to solve this problem in the area of program specialization, but it has not been created for relational programming before. In this paper we propose a binding-time analysis algorithm for the relational programming language miniKanren.

Index Terms—Relational programming, binding-time analysis, static analysis