Анализ времени связывания для реляционных программ

1st Ирина Артемьева
Университет ИТМО
Санкт-Перебург, Россия
irinapluralia@gmail.com

2nd Екатерина Вербицкая
JetBrains Research
Санкт-Перебург, 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.

Index Terms—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 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 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. В статье показано, что специализация приводит к су-

[1] Официальный сайт языка Mercury: http://mercurylang.org/, дата последнего посещения: 11.02.2020

[2] Официальный сайт языка miniKanren: http://minikanren.org/, дата последнего посещения: 11.02.2020

щественному 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 разниться 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 <u>fresh</u>.

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

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

```
appendo x y xy =
    (x == [] /\ y == xy) \/
    (fresh [h, t, r] (
        x == h : t  /\
        xy == h : r /\
        invoke "appendo" [t, y, r]
    ))
```

Исполнение этого отношения в прямом направлении на двух заданных списках $append^o$ $[1, 2]$ $[3]$? 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 $append^o$? ? $[1, 2, 3]$ является множество пар $\{([], [1, 2, 3]), ([1], [2, 3]), ([1, 2], [3]), ([1, 2, 3], [])\}$.

III. Трансляция в функциональный язык

Рассмотрим схему трансляции miniKanrenв Haskellв первом приближении. Отношение с выбранным направлением соответствует одной функции. Каждая дизъюнкция - новое уравнение этой функции. Конъюнкция представляет собой или унификацию, или вызов другого отношения. При трансляции все вызовы превращаются в определения переменных, а унификации - либо так же в определения переменных, либо становятся переменными сопоставления с образцом для соответствующего уравнения. В результате трансляции ожидается получить функцию на Haskell, семантика вычисления которой соответствует семантике вычисления отношения в выбранном направлении на miniKanren. Так же ожидается, что время выполнения вычислений на Haskell будет ощутимо меньше времени выполнения на miniKanren. Рассмотрим примеры трансляции *append^o* в различных направлениях.

При прямом направлении первый и второй аргументы — входные, третий — выходной. Здесь два дизъюнкта, поэтому функция на Haskell состоит из двух уравнений. В первом дизъюнкте ?? два конъюнкта-унификации. Первый становится сопоставлением с образцом первого аргумента, а второй - определением, которое позволяет получить значение третьего аргумента. Во втором дизъюнкте три конъюнкта. Первый ?? — унификация, первого аргумента — становится сопоставлением с образцом. Второй ?? — так же унификация — становится определением третьего аргумента. Третий ?? — вызов отношения *appned^o* — должен стать определением. На данном этапе необходимо определить подходящее направление вычисления вызываемого отношения. Так как мы знаем, что данный вызов рекурсивный — будем считать, что направление соответствует направлению при трансляции. Пример соответствующей трансляции в язык Haskell приведён ниже.

```
appendo [] y = xy
  where
    xy = y
appendo (h : t) y = xy
  where
    r  = appendo t y
    xy = h : r
```

Транслируем в обратном направлении: третий аргумент — входной, первый и второй — выходные. Для данного отношения два выходных аргумента гарантируют недетерминированность. Трансляция первого дизъюнкта тривиальна — для возвращения нескольких переменных будем использовать кортеж; недетерминированности в этом дизъюнкте нет. Во втором дизъюнкте есть рекурсивный вызов - по нашей эвристике его направление совпадает с направлением при трансляции. По семантике вызова *append^o* на таком направлении вернётся список пар списков, конкатенация которых даст исходный список — недетерминированность. Для её поддержки сделано следующее:

- вычисление результата вызова отношения в монаде списка;
- каждое уравнение функции теперь отдельная функция, возвращающая пустой список в случае неудачи;
- результаты всех уравнений-функций объедим при помощи конкатенации в функции на верхнем уровне;
- рекурсивные вызовы внутри уравнений-функций относятся к функции на верхнем уровне (в примере ниже *append*^{0}2 вызывает *append^o*).

```
appendo x = appendo1 x ++ appendo2 x
  where
    appendo1 y = do
      let x = []
      return (x, y)
    appendo1 _ = []

    appendo2 (h : r) = do
      (t, y) <- appendo r
      let x = h : t
      return (x, y)
    appendo2 _ = []
```

Анализируя примеры трансляции, можно сделать несколько выводов.

Первый. Направление вычисления отношения влияет на порядок вычислений внутри этого отношения. При вычислении второго дизъюнкта в прямом направлении конъюнкты вычисляются в порядке 4 6 5, а в обратном направлении — 5 6 4.

Второй. Направление вычисления отношения влияет на выбор направления вычисления конъюнктов (унификаций и вызов отношения). При трансляции первого дизъюнкта в прямом направлении унификация x и пустого списка в 2 уходит в сопоставление с образцом, где происходит попытка присвоения x пустому списку. В обратном направлении унификация x присваивается пустой список. Рекурсивные вызовы в примерах выше также происходят в разных направлениях. При прямом порядке третий аргумент — выходной 6, а при обратном — входной 9. В примерах для определения направления вызываемых отношений достаточно использовать эвристику, так как вызовы рекурсивны. На практике при трансляции необходимо определять как направление вычисления вызываемых отношений, так и унификаций.

Третий. Использование монад накладывает ограничение на порядок определения переменных. В последнем примере нельзя поменять местами 9 и 10 в функции *append*^{0}2.

ИСТОРИЧЕСКАЯ СПРАВКА О BTA

Анализ времени связывания решает обе эти проблемы: аннотируя каждую переменную цели временем связывания, можно выбрать направление вычислений внутренних целей, и по аннотациям восстановить верный порядок определений.

IV. Анализ времени связывания для miniKanren

Алгоритм получает на вход цель, данные о входных переменных и каждой переменной цели ставит в соответствие число. Процесс подбора чисел называется аннотированием.

Инициализация алгоритма состоит из двух частей:

- уникально переименовать все $fresh$-переменные, чтобы избежать перекрытия имён;

- нормализовать (привести к дизъюнктивной нормальной форме) для упрощения алгоритма;

Опишем шаг алгоритма.

—————————————————————————————

Анализ времени связывания - это статический анализ, использующий монотонный фреймворк [7] — тройку из полурешётки L, $meet$-операции и множества монотонных функций F, ассоциированных с конкретными экземплярами полурешётки L, и удовлетворяющих свойству монотонности.

В нашем случае элементы полурешётки L — аннотации. Аннотация может быть или $Undef$ (наименьший элемент) для случая, когда о переменной ничего не известно, или целое число — время связывания переменной. На целых числах соблюдается естественный порядок, а $Undef$ считается меньше численной аннотации.

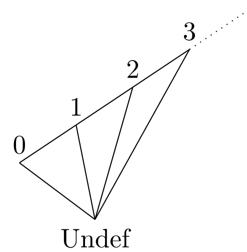

Рис. 1. Полурешётка

Операция $meet$ устроена так, чтобы обеспечивать монотонность: переменная, проаннотированная значением n, никогда не будет проаннотирована значением, меньшим n. Таким образом, из $Undef$ аннотации можно перейти в любую численную аннотацию, а из численной аннотации можно перейти в численную аннотацию не меньшую текущей.

Цель анализа времени связывания — указать порядок, в котором имена связываются со значениями. Когда алгоритм принимает цель с указанием направления, он выполняет первичное аннотирование — входные аргументы считаются известными в момент времени 0, поэтому получают аннотацию 0, остальные аргументы получают аннотацию $Undef$. Переменные, проаннотированные числами, считаются известными или константами к текущему моменту времени, поэтому $Undef$-переменные, зависящие только от проаннотированных переменных, могут получить свою аннотацию, значение которой будет больше на 1 самого большого значения аннотации переменных, от которых она зависит. Таким образом информация о времени связывания может распространяться на другие переменные.

Реализация разработанного алгоритма доступна на GitHub[3]. Опишем работу алгоритма.

Входные и выходные данные

- принимает программу на miniKanren и список входных переменных
- возвращает пару из проаннотированной нормализованной цели (приведённой к дизъюнктивной нормальной форме) и стека вызовов (отображения из названия отношения во множество информации о будущей функции: направление вычисления отношения и цель, размеченная по этому направлению)

Инициализация цели перед аннотированием

- снять все fresh, дав переменным уникальные имена
- нормализовать
- произвести первичное аннотирование цели данными о входных переменных

Аннотирование

- создать пустой стек вызовов
- до fix point — вычисляем аннотации нормализованной цели, пока не достигнем неподвижной точки
- аннотировать нормализованную цель — аннотировать все её дизъюнкты
- аннотировать дизъюнкт — аннотировать все его конъюнкты (последовательно передавая стек вызовов), а затем распространить информацию об аннотировании между конъюнктами: для каждой переменной дизъюнкта получить её аннотации из всех конъюнктов, найти максимальную и установить её значение в качестве аннотации этой переменной во всём дизъюнкте
- аннотировать конъюнкт — аннотировать либо унификацию, либо вызов отношения
- аннотировать унификацию
  - слева переменная с $Undef$-аннотацией — получить максимальную аннотацию правого терма, увеличить её на 1 и присвоить аннотации левого терма
  - слева переменная, аннотированная числом — увеличить её значение на 1 и установить в качестве значения всех $Undef$-аннотаций правого терма
  - слева и справа конструкторы — произвести zip аргументов и вызвать аннотацию аргументов для каждой пары; полученные унификации разбить на два списка аргументов конструкторов
  - оставшиеся случаи зеркальны и обрабатываются аналогично
- аннотировать вызов отношения:
  - если все термы вызова проаннотированы $Undef$ или все проаннотированы числами, вернуть исходную цель
  - если вызов с таким именем и направлением есть в стеке вызовов, определить максимальную аннотацию аргументов вызова, неравную $Undef$, увеличить её на 1 и проаннотировать её значением переменные с $Undef$-аннотацией

[3]github.com/Pluralia/uKanren_translator

– если в стеке нет вызова с таким именем и направлением, добавить его и текущее направление в стек вызовов, проаннотировать цель, полученную по имени, с учётом текущего направления и обновлённого стека вызовов, обновить стек ещё раз: данному вызову и данному направлению доавить проаннотированную цель

Пример работы - отношение $revers^o$

$$
\begin{aligned}
&revers^o\ x\ y = \\
&\qquad (x \equiv [] \wedge y \equiv []) \vee \\
&\qquad (\underline{fresh}\ [h, t, r](\\
&\qquad\ x \equiv h : t \wedge \\
&\qquad\ \underline{call}\ revers^o\ t\ r \wedge \\
&\qquad\ \underline{call}\ append^o\ r\ [h]\ y))
\end{aligned}
$$

Рассмотрим его аннотирование в направлении y - входная переменная

Пример работы - проаннотируем $append^o$

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. Вопрос об эффективном и корректном способе обработки таких ситуаций на данный момент остается открытым.

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 programmingproblems (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] J. B. Kam and J. D. Ullman, “Monotone data flow analysis frameworks,” in Acta Informatica. Springer, 1977, pp. 305–317.