Анализ времени связывания для реляционных программ

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

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.**

***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 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.

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

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

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 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 fresh.

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

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 $x_0, x_1, \ldots, x_n$.

```
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 $append^o$ $[1, 2]$ $[3]$? 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

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 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] → [a] → [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 $R \subseteq X_0 \times \cdots \times X_n$, в котором известны аргументы $X_{i_0}, \ldots X_{i_k}$, а аргументы $X_{j_0}, \ldots X_{j_l}$ необходимо вычислить, соответствует функция $F : X_{i_0} \to \cdots \to X_{i_k} \to [X_{j_0} \times \cdots \times X_{j_l}]$, 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:

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

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 в 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[3]. Связывание в строке 10 означает, что результат будет вычислен для каждого элемента списка, полученного из рекурсивного вызова функции.

```
1  append^o :: [a] → [([a], [a])]
2  append^o x = append^o_1 x ++ append^o_2 x
3    where
4      append^o_1 y = do
5        let x = []
6        return (x, y)
7      append^o_1 _ = []
8
9      append^o_2 (h : r) = do
10       (t, y) ← append^o r
11       let x = h : t
12       return (x, y)
13     append^o_2 _ = []
```

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.

[3]Описание do-нотации языка Haskell: https://en.wikibooks.org/wiki/Haskell/do_notation, дата последнего посещения: 14.02.2020

```
1  revers^o xs sx =
2    (xs ≡ [] ∧ sx ≡ []) ∨
3    (fresh [h, ts, st] (
4        x ≡ h : ts ∧
5        revers^o ts st ∧
6        append^o [st, [h], sx]
7    ))
```

Рис. 4. Отношение $revers^o$

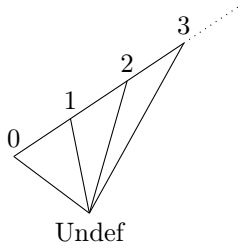

Рис. 5. Полурешётка

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

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.

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[4]. 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

[4]github.com/Pluralia/uKanren_translator

- 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
  - 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 $revers^o$

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

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.