

Анализ времени связывания для реляционных программ

1st Ирина Артемьева
dept. name of organization (of Aff.)
name of organization (of Aff.)
Saint-Petersburg, Russia
email address or ORCID

2nd Екатерина Вербицкая
JetBrains Research
Санкт-Петербург
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

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. Выполнение отношения $add^o \ x \ y \ ?$ с зафиксированными 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. Основным представителем парадигмы реляционного программирования является семейство интерпретируемых языков miniKanren. 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. 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 [1]. Так, имея интерпретатор языка, можно решать задачу синтеза программ на этом языке по набору тестов [?]; 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 [1] (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 [1] для решения этой проблемы используется специализация. В статье показано, что специализация приводит к существенному приросту скорости работы программы. Однако чтобы избавиться от всех накладных расходов, связанных с интерпретацией программы, необходим Джонс-оптимальный специализатор [2]. К сожалению, реализация такого специализатора — нетривиальная задача.

В данное время авторами ведется работа над аль-

тернативным 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 [3], 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, схему его трансляции в функциональный язык и возникающие при этом трудности. Алгоритм анализа времени связывания для miniKanren приведен в разделе III. В заключении (раздел IV) мы подводим выводы и описываем планы на дальнейшую работу.

II. Язык программирования miniKanren

miniKanren- особенный язык в мире реляционного программирования: его минималистичность и чистота породили интерес ко всей парадигме. Рассмотрим его подробнее: конструкции и примеры исполнения при выборе различных направлений.

Первая конструкция называется термом - это либо переменная, либо конструктор, содержащий кортеж термов.

Любое отношение состоит из трёх частей: имя, имена аргументов (аналогично объявлению свежих переменных) и цель.

Следующая конструкция - цель. Целью может быть унификация двух термов, дизъюнкция и конъюнкция двух целей, вызов какого-то другого отношения по имени с передачей аргументов (в качестве которых выступают термы); перед использованием ранее неизвестных переменных внутри цели их нужно "объявить"аналогично свежим переменным в лямбда-исчислении - такая конструкция так же является целью.

Программа на miniKanren- список отношений (может быть пустым) и цель, которая может вызывать эти отношения.

Рассмотрим простой пример. Пусть у нас есть отношение (указано без цели; его семантика указана в фигурных скобках):

$$\text{sum } x \ y \ z = \{ z \mid x + y == z \}$$

В зависимости от выбора направления вычислений мы можем получить из этого отношения следующие функции (выходные аргументы обозначены ?):

- $\text{add } a \ b = \text{sum } a \ b \ ?$
- $\text{double } a = \text{sum } a \ a \ ?$
- $\text{subtract } a \ b = \text{sum } ? \ b \ a$
- $\text{addends } a = \text{sum } ? \ ? \ a$

addends здесь выступает примером обратного вычисления функции: она возвращает все возможные пары чисел, сумма которых может дать *a*. Отдельно стоит отметить, что в данном случае мы получили недетерминированный ответ: могут существовать несколько пар чисел, сумма которых будет давать исходное число.

Перейдём к особенностям трансляции в функциональный язык. Как уже было сказано, выбор направления вычисления отношения влияет на порядок вычислений внутри этого отношения. Это означает, что при трансляции одного и того же отношения в разных направлениях цели внутри этих отношений будут вычисляться в разном порядке. Тем не менее, мы знаем, что в Haskell есть *let* и *where*, внутри которых порядок объявлений не важен. Почему мы не можем их использовать и забыть об анализе времени связывания?

Проблема возникает по двум причинам. Первая проблема - необходимо поддерживать недетерминированность (пример с *addends*), что накладывает ограничения на использование *let* и *where*. Вторая проблема - для задания объявления по отношению необходимо определять их направления. Рассмотрим подробнее каждую из проблем.

Недетерминированность - одна из вещей, которую необходимо сохранить при трансляции в функциональный язык. В Haskell есть монада списка, которая идеально позволяет справиться с возникшей проблемой: будем использовать *do*-нотацию и, если необходимо обработать унификацию - пишем *let*, а, если вызов отношения (вернёт список), то развернём как монаду. В конце все ответы соберём в кортеж и сделаем *return*, запаковав всё обратно в списковую монаду. Это способ решения проблемы поддержания недетерминированности, но проблема в том, что в *do*-нотации порядок объявлений важен - их нужно упорядочивать.

Касательно выбора направления объявлений - поймём, из чего происходят объявления: существует два типа целей, которые связывают переменные: унификация и вызов другого отношения. Ни тот, ни другой тип цели, очевидно, не имеют направления вычисления в miniKanren. Это значит, что его нужно определять в процессе, "на лету по данным о направлении вычисления исходного отношения. Для унификации необходимо понять, левая часть присваивается правой или наоборот, а для вызовов функций нужно научиться определять направление.

Анализ времени связывания решает обе эти проблемы: аннотируя каждую переменную цели временем связывания, можно выбрать направление вычисления

внутренних целей, и по аннотациям восстановить верный порядок объявлений.

III. Анализ времени связывания для miniKanren

При реализации алгоритма анализа времени связывания были использованы идеи монотонного фреймворка [4]. Другими словами, то, что необходимо сделать для

Алгоритм

Общее описание - принимает программу и список input-переменных - возвращает пару из проаннотированной нормализованной цели и стека (мапа из названия отношения во множество информации о будущей функции; эта информация включает направление вычисления отношения и цель, размеченную по этому направлению)

translate 1. Получаем gamma 2. Инициализируем goal - снимаем все fresh и переименовываем переменные - нормализуем - аннотируем инфой об аргументах, а также инфой о константах 3. Запускаем аннотирование ('annotate')

annotate / annotateInternal 4. Добавляем к цели пустой стек 5. До fix point (пара из цели и стека не изменилась с предыдущего вызова) будем вызывать 'annotateGoal'

annotateGoal 6. Последовательно пройдемся по всем дизъюнктам: 'annotateDisj' передаём текущий disj и стек после вычислений всех предыдущих disjs

annotateDisj 7. То же, что и в 'annotateGoal', но обходим конъюнкты (функция 'annotateConj') - в конце делаем 'meetGoal' между всеми конъюнктами

meetGoal - принимает список конъюнктов, получаем все-все переменные из них, сортирует, группирует - получаем переменную и её аннотации со всех конъюнктов; для каждой переменной на всех её аннотациях вызываем 'meet'

meet - одна из переменных Undef - возвращаем другую - обе чем-то проаннотированы - возвращаем максимум

annotateConj 8. на данном этапе цель - это либо унификация, либо инвок; разберём обработку каждого из случаев отдельно

Унификация - на термах вызываем 'meetTerm'

meetTerm - слева undef-переменная - получаем максимальную аннотацию правого терма (вызываем 'maxAnn') - слева аннотированная переменная - увеличим её значение на 1 и поставим везде вместо всех Undef в правом терме (используем функцию 'replaceUndef' - она ругается на нарушение монотонности, если у внутри терма нашлась переменная, аннотация которой оказалась больше той, на которую хотим заменить) - слева и справа конъюнкты - проверяем, что их имена и кол-во аргументов совпадают, затем делаем zip аргументов и вызываем 'meetTerm' на каждой паре аргументов - оставшиеся случаи обрабатываются вы-

зовом 'meetTerm' на аргументах в обратном порядке; в конце восстанавливается исходный порядок

maxAnn - переменная - очевидно - конструктор - если есть хотя бы 1 внутренний Undef терм, вернём Undef, иначе вернём максимум из аннотаций

Инвоки - существует 3 случая: 1. все термы Undef - ничего не делаем (будем ждать следующую итерацию, когда о термах станет известно хоть что-то) 2. такая цель уже есть в стеке (проверяем по имени и маске аннотаций термов - список списков, в котором n-ый список содержит номера переменных, инициализируемых в n-ый момент времени) - доопределяем Undef-термы как максимальная аннотация оставшихся термов + 1 - добавляем в стек инфу о новом направлении (ВАЖНО: в качестве цели здесь кладётся undefined - может вылезти боком), при этом происходит сравнение масками с инфой об уже существующих направлениях: оставляется то направление, которое даёт больше информации о времени инициализации термов (пример: пусть есть маски [[2], [0, 1]] и [[2], [1], [0]]; они не противоречат друг другу и вторая даёт больше информации, поэтому её направление "затрёт" в стеке направление первой). 3. цели в списке не оказалось - получаем def по имени (из gamma) - по def получаем без фрейшей, переименованную, нормализованную, первично проинициализированную (инфой об аннотациях термов при вызове) цель, список ПЕРЕИМЕНОВАННЫХ названий ЕЁ входных переменных, а также обновляем стек, складывая туда текущую маску и цель - аннотируем полученную цель, вызывая 'annotateInternal' - получаем новую цель и новый стек - из цели получаем аннотации её аргументов и переносим эти аннотации на термы при инвоке - обновляем инвок - в уже изменённый стек по маске изменённых на прошлом шаге термов добавляем полученную цель

Вот поэтому считаем, что это работает. Вот тут семантика для miniKanren [5].

IV. Заключение

Таким образом, мы разработали анализ времени связывания для miniKanren, а также доказали, что он работает. Основным его узким местом является случай вызова нескольких отношений на переменных, которые используются только в этих вызовах - в этом случае приходится перебирать все возможные сочетания направлений трансляции этих отношений, что может быть довольно долго или давать неправильный ответ. На текущий момент вопрос о способе обработки такого рода случаев остаётся открытым.

Так же в дальнейшем будет продолжена работа над транслятором в функциональный язык - анализ времени связывания является ядром этого транслятора. По проаннотированной программе предстоит получить объявления в корректном порядке, а также продумать и реализовать корректную трансляцию остальных типов целей: дизъюнкций, конъюнкций.

Acknowledgment

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

Список литературы

- [1] P. Lozov, E. Verbitskaia, and D. Boulytchev, “Relational interpreters for search problems,” in Relational Programming Workshop, 2019, p. 43.
- [2] N. D. Jones, C. K. Gomard, and P. Sestoft, Partial evaluation and automatic program generation. Peter Sestoft, 1993.
- [3] W. Vanhoof, M. Bruynooghe, and M. Leuschel, “Binding-time analysis for mercury,” in Program Development in Computational Logic. Springer, 2004, pp. 189–232.
- [4] J. B. Kam and J. D. Ullman, “Monotone data flow analysis frameworks,” in Acta Informatica. Springer, 1977, pp. 305–317.
- [5] D. Rozplokhos, A. Vyatkin, and D. Boulytchev, “Certified semantics for minikanren,” in and Relational Programming Workshop, 2019, p. 80.