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

Аннотация—

Index Terms—Реляционное программирование, анализ времени связывания

I. Введение

Реляционное программирование — парадигма, в которой любая программа рассматривается как математическое отношение. Это значит, что аргументы такой программы, своего рода, равноправны и, чтобы получить какой-то результат, необходимо указать, какие из аргументов являются входными, а какие - выходными. В данной статье будем работать с реляционным программированием на примере miniKanren.

Процедура специфицирования аргументов называется выбором направления вычисления. Два основных преимущества реляционного программирования кроются именно в этом. Во-первых, выбор направления позволяет переиспользовать один и тот же код (конкретное отношение) для вычисления различных функций. Так, например, любой аргумент можно обозначить как входным, так и выходным. Во-вторых, появляется возможность вычислять функции в обратном направлении. Отношение - это не функция; пусть мы задали отношение на аргументах и результате функции - теперь мы можем выбирать направление вычислений: можем, как и раньше, получать результат по аргументам, а можем, - все возможные значения аргументов по результату.

Вычисление функций в обратном направлении позволяет, например, создать интерпретатор языка и использовать его для генерации программ по набору тестов [?] или решать задачи поиска силами распознавателя. [1]

Стоит отметить, частично такими преимуществами обладает и логическое программирование, но в нём есть нереляционные вещи: способ обхода пространства решений (которое может быть весьма большим) требует искусственного уменьшения этого пространства - иногда приходиться выбирать направление вычисления внутри программы, из-за чего такая программа больше не является отношением в чистом виде, а соответствующий язык - чистым логическим языком.

Реляционный язык можно назвать чистым логическим языком. Его отличие - мы никогда не выбираем направление внутри программы. Тем не менее, проблема с необходимостью обходить большое пространство решений остаётся - всё это влияет на скорость поиска решения. В [2] показано, что даже имея хороший специализатор, сложно достичь быстрого вычисления реляционных программ, поэтому в данной работе мы предлагаем другой подход - транслировать реляционную программу в функциональную, предварительно выбрав направление вычисления отношения. В качестве конкретного функционального языка выберем .

Выбор направления вычисления отношения влияет на порядок вычислений внутри этого отношения, и устройство реляционного языка позволяет не обращать на это внимание, однако, при трансляции реляционной программы в функциональную эта проблема появляется: функциональным языкам важен порядок связывания имён в программах, а, значит, для успешной трансляции его нужно определить. Для этих целей был создан анализ времени связывания. В mercury(функционально-логический язык программирования) этот подход уже применяется [3], но для реляционных языков подобного нет.

Оставшаяся часть статьи построена следующим образом: в секции 2 - обзор конкретного реляционного языка miniKanren, обсуждение проблем, связанных с его трансляцией в функциональный язык, и обоснование использования анализа времени связывание; в секции 3 - описание анализа времени связывания для miniKanreni обоснование работоспособности предлагаемого подхода; в секции 4 - выводы о полученных результатах и описание дальнейшей работы.

II. 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):

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

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

- $add\ a\ b\ =\ sum\ a\ b\ ?$
- $double\ a\ =\ sum\ a\ a\ ?$
- $substract\ a\ b\ =\ sum\ ?\ b\ a$
- $addends\ a\ =\ sum\ ?\ ?\ a$

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 недетерменированный 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 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 недетерменированность (пример с addends), что накладывает ограничения на использование let и where. Вторая проблема - для задания объявления по отношениям необходимо определять их направления. Рассмотрим подробнее каждую из проблем.

Недетерменированность - одна из вещей, которую необходимо сохранить при трансляции в функциональный язык. В 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 недетерменированности, 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 проаннатированной 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 пройдёмся 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 - вовзращаем 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 анноцию правого терма (вызываем ‘maxAnn‘) - слева аннотированная переменная - увеличим её значение на 1 и проставим 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 кторы - проверяем, что их именна и кол-во аргументов совпадают, затем делаем zip аргументов и вызываем ‘meetTerm‘ на каждой паре аргументов - оставшиеся случаи обрабатываются вызовом ‘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 следущую 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 фрешей, переименованную, нормализованную, первично проинициализированную (инфой об аннотациях термов при вызове) цель, список ПЕРЕИМЕННОВЫННЫХ 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 приходиться 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: дизъюнкции, конъюнкции.

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. Rozplokhas, A. Vyatkin, and D. Boulytchev, “Certified semantics for minikanren,” in and Relational Programming Workshop, 2019, p. 80.