Санкт-Петербургский государственный университет

Кафедра системного программирования

Группа 21.Б10-мм

Специализация реляционных программ

Ерин Игорь Антонович

Отчёт по учебной практике
в форме «Производственное задание»

Научный руководитель:
ассистент кафедры системного программирования, Косарев Д. С.

Санкт-Петербург
2023

Оглавление

Введение

Реляционное программирование — это парадигма, основанная на выражении программ с помощью отношений. Отношения сами собой представляют функции, но, в отличие от функционального программирования, исполнять их можно в различных направлениях. Это позволяет естественно выражать некоторые проблемы [9], среди которых генерация программ посредством написания реляционных интерпретаторов [2].

Повышение абстракции зачастую приводит к худшей производительности. Не стали исключением и реляционные языки программирования [1, 10].

На листинге 3 изображено отношение меньше или равно.

Листинг 1: Отношение меньше или равно

```
let rec is_le x y is =
    conde
      [ x ≡ O &&& (is ≡ true)
      ; x ≢ O &&& (y ≡ O) &&& (false ≡ is)
      ; fresh (x' y')
          x ≡ S x'
          &&& (y ≡ S y')
          &&& le x' y' is
      ]
```

Иногда пользователя интересуют те случаи, когда некоторые аргументы известны до исполнения. К примеру, в случае упомянутого выше отношения, обнаружить пары чисел, для которых это отношение выполняется.

Листинг 2: Отношение меньше или равно

```
let le x y = is_le x y true
```

Реляционное программирование предоставляет бóльшую гибкость, так как заменив значение одного аргумента позволяет определить обратное отношение.

Листинг 3: Отношение меньше или равно

```
let gt x y = is_le x y false
```

Возможно, именно эта гибкость становится преградой, что скрывает за собой производительность. Однако, до исполнения можно произвести подстановку определенного аргумента в тело функции, дабы частично вычислить ее, отбросить все лишнее, тем самым получить, возможно, более производительную версию.

Отношения содержащие параметры с конечным доменом, например, имеющие тип `bool`, представляют особенный интерес. Так как в зависимости от того, какое значение будет использовано в качестве аргумента, `true` или `false`, при интерпретации могут быть задействованы совершенно разные части реляционной формулы.

Таким образом, данная работа посвящена исследованию вопроса подстановки и специализации реляционной программы для аргументов с конечным доменом.

Дата сборки: 12 декабря 2023 г.

1. Постановка задачи

Целью данной работы является реализация специализатора реляционных программ написанных на языке OCanren[1], диалекте miniKanren[2]. Для этого были поставлены следующие задачи.

- Реализовать специализатор.
- Сравнить производительность специализированных и исходных функций.

[1]Репозиторий проекта OCanren: https://github.com/PLTools/OCanren (Дата обращения: 10.12.2023)

[2]Сайт языка miniKanren: http://minikanren.org/ (Дата обращения: 10.12.2023)

2. Обзор

В данном разделе приведен краткий обзор подходов к специализации программ.

2.1. Подходы к специализации

Известны множество подходов к специализации императивных, функциональных и логических языков. Такие методы, как частичные вычисления [4], суперкомпиляция [8], дистилляция [3] и частичная дедукция [6].

Все эти подходы объединяет символическое исполнение обрабатываемой программы, называемое driving, в процессе которого строится так называемое processing tree, потенциально бесконечное, что должно отразить саму сущность программы [8]. В процессе построения термы, расположенные в узлах дерева подвергаются проверкам, направленным на установление расхождения, например, с помощью [5]. После чего, по полученной структуре генерируется результирующая программа.

Реляционное программирование отлично от логического полнотой поиска [1]. В настоящий момент техники специализации, основанные на оных для логических языков, разрабатываются для реляционных программ [10].

3. Реализация

В данном разделе описаны подходы к реализации.

3.1. Исходное представление

Так как специализируемым языком выступал OCanren, встроенный в OCaml, в качестве исходного представления программ на стадии проектирования предполагалось использовать одно из представлений, которое порождает компилятор OCaml. Таковым было выбрано типизированное дерево, ибо типы необходимы для установления конечности домена и генерации всех возможных значений.

3.2. Промежуточное представление

В качестве промежуточного представления была выбрана дизъюнктивная нормальная форма (далее ДНФ). Ибо она позволяет рассматривать каждый конъюнкт независимо от других. Что в свою очередь упрощает протягивание констант и редукцию всей формулы.

3.3. Специализация

Рассмотрим редукцию следующей формулы.

Листинг 4: Отношение вычитания

```
let sub x y z =
   fresh (valid)
     loe y x valid
     &&& conde
           [ valid ≡ false &&& (z ≡ None)
           ; fresh (z_value)
               valid ≡ true
               &&& (z ≡ Some z_value)
               &&& add y z_value x)
           ]
```

Специализация будет происходить по параметру `z` и конструктору `Some`. Необходимо помнить, что арность данного конструктора равна единице.

Сначала формула будет приведена в дизъюнктивную нормальную форму. Объявление свежих переменных необходимо переместить, чтобы их область видимости состояла из всего конъюнкта, при необходимости переименовав.

Листинг 5: Отношение в ДНФ

```
let sub x y z =
   fresh (valid) (loe y x valid) (valid ≡ false) (z ≡ None)
   ||| fresh (valid z_value)
         (loe y x valid)
         (valid ≡ true)
         (z ≡ Some z_value)
         (add y z_value x)
```

Редуцируемый параметр будет заменен на параметры конструктора, в данном случае `some_arg`. Все вхождения редуцируемого параметра будут заменены на конструктор с новым параметром в качестве аргумента.

Листинг 6: Отношение после подстановки конструктора

```
let sub x y some_arg =
   fresh (valid)
       (loe y x valid)
       (valid ≡ false)
       (Some some_arg ≡ None)
   ||| fresh (valid z_value)
         (loe y x valid)
         (valid ≡ true)
         (Some some_arg ≡ Some z_value)
         (add y z_value x)
```

Теперь необходимо протянуть константы.

3.3.1. Протягивание констант

Все имена, к которым применялось отношение унификации будут разбиты на классы эквивалентности. После чего в каждом классе эквивалентности будет выбран представитель. Затем все вхождения свободных переменных этого класса будут заменены на данного представителя.

Листинг 7: Отношение после протягивания констант

```
let sub x y some_arg =
   fresh (valid)
       (loe y x false)
       (false ≡ false)
       (Some some_arg ≡ None)
   ||| fresh (valid z_value)
         (loe y x true)
         (true ≡ true)
         (Some some_arg ≡ Some z_value)
         (add y z_value x)
```

3.3.2. Редукция

Во время редукции производится проверка на выполнимость. Конъюнкты содержащие заведомо невыполнимую унификацию будут удалены. Избыточные конструкторы будут сняты.

Листинг 8: Отношение после редукции

```
let sub x y some_arg =
   fresh (valid z_value)
         (loe y x true)
         (some_arg ≡ z_value)
         (add y z_value x)
```

Протягивание констант и редукцию необходимо повторять до схождения к неподвижной точке.

Листинг 9: Отношение с лишними свежими именами

```
let sub x y some_arg =
   fresh (valid z_value)  (loe y x true) (add y some_arg x)
```

После чего необходимо в каждом конъюнкте независимо удалить неиспользуемые свежие переменные.

Листинг 10: Результирующее отношение

```
let sub x y some_arg =
   fresh ()  (loe y x true) (add y some_arg x)
```

3.3.3. Замыкание

В результате может получиться формула, в теле которой окажутся вызовы отношений с известными параметрами. Такие отношения так же необходимо специализировать, их специализированный вызовы подставить в текущую формулу. Осуществляется это с помощью обхода графа вызовов, начиная с исходной функции.

Листинг 11: Отношение со специализированным вызовом

```
let sub x y some_arg =
   fresh ()  (loe_true y x) (add y some_arg x)
```

3.4. Трансляция

По редуцированной формуле строится нетипизированное дерево. Затем средствами стандартной библиотеки компилятора оно транслируется в код языка OCaml.

4. Эксперимент

В данном разделе приведены условия сравнительных экспериментов и их результаты.

4.1. Цель эксперимента

Целью эксперимента является сравнение производительности специализированных и исходных функций.

4.2. Условия эксперимента

Для эксперимента были выбраны отношения: is_even[3], sub[4], gcw[5], bridge[6].

Эксперименты проведены на машине, имеющей следующие характеристики: Ubuntu 20.4, AMD Ryzen 5 5500U, 4.4GHz, DDR4 16GB RAM. Для измерения был использован ocaml-benchmark[7].

4.3. Результаты эксперимента

Результаты измерений приведены в следующих таблицах, где `x` и `spec_x` обозначают соответственно результаты исполнения функции с конструктором `x` и исполнение специализированной по этому конструктору функции. В столбце `Частота` обозначено количество исполнений за секунду. Большее — лучше. Во всех случаях отклонение составило менее 3%, вследствие чего оно не приводится. Каждый эксперимент состоял из 30 замеров.

[3]Код отношения is_even https://github.com/IgorErin/SpecialKanren/blob/master/samples/is_even.ml (Дата обращения 10.12.2023)

[4]Код отношения sub : https://github.com/IgorErin/SpecialKanren/blob/master/samples/sub.ml (Дата обращения 10.12.2023)

[5]Код отношения gcw: https://github.com/IgorErin/SpecialKanren/blob/master/samples/gcw.ml (Дата обращения 10.12.2023)

[6]Исходный код отношения bridge https://github.com/IgorErin/SpecialKanren/blob/master/samples/bridge.ml (Дата обращения 10.12.2023)

[7]Репозиторий ocaml-benchmark: https://github.com/Chris00/ocaml-benchmark (Дата обращение 10.12.2023)

Таблица 1: Результаты измерений is_even при получении первых ста ответов.

Название	Частота	false	spec_false
false	485	–	-30%
spec_false	695	43%	–

Таблица 2: Результаты измерений is_even при получении первых ста ответов.

Название	Частота	true	spec_true
true	486	–	-29%
spec_true	688	42%	–

Таблица 3: Результаты измерений sub при получении первых двадцати пяти ответов.

Название	Частота	none	spec_none
none	5598	–	-24%
spec_none	7329	31%	–

Таблица 4: Результаты измерений sub при получении первых двадцати пяти ответов.

Название	Частота	some	spec_some
some	1071	–	-4%
spec_some	1113	4%	–

Таблица 5: Результаты измерений gcw при получение первых ста ответов.

Нзавние	Частота	spec_false	false
spec_false	366	–	-19%
false	454	24%	–

Таблица 6: Результаты измерений gcw при получении первых ста ответов.

Название	Частота	true	spec_true
true	1.25	–	-52%
spec_true	2.61	108%	–

Таблица 7: Результаты измерений bridge при получении первых ста ответов.

Название	Частота	none	spec_none
none	421	–	-29%
spec_none	596	42%	–

Ввиду продолжительности исполнения отношения bridge, приведена величина, обратная `Частоте`. То есть в столбце `Скорость` обозначено среднее количество секунд за исполнение. Меньшее — лучше.

Таблица 8: Результаты измерений bridge при получении первого ответа.

Нзавние	Скорость	some	spec_some
some	15.8	–	-43%
spec_some	9.03	75%	–

4.4. Выводы

Из приведенных результатов видно, что специализированные версии эффективнее всегда, за исключением эксперимента 5, где проигрыш может быть объяснен большим количеством дублицированного кода вследствие использования ДНФ. Во многих конъюнктах создаются большое количество свежих имен, которые в исходной формуле аллоцировались единожды. Данный случай показывает необходимость слияния одинаковых частей конъюнктов, по крайней мере аллокации переменных.

Заключение

В ходе данной работы были выполнены следующие задачи.

- Реализован простой специализатор реляционных программ.

  Код работы доступен в репозитории на сервисе GitHub[8]. Имя пользователя: IgorErin.

- Произведено сравнение изменения производительности некоторых функций до специализации и после.

Даже такая простая специализация способна значительно улучшить время исполнения реляционных программ.

Далее предполагается:

- Увеличить класс обрабатываемых программ. Добавить поддержку модулей, частичного применения и тому подобного.

- Улучшить качество генерируемого кода. Уменьшить количество дублицируемых конструкций, поддержать инлайнинг простых реляций.

После чего необходимо будет провести эксперимент на настоящих программных продуктах, активно использующих реляционное программирование, например [7].

[8]Репозиторий проекта OCanren: https://github.com/IgorErin/SpecialKanren (Дата обращения: 10.12.2023)

Список литературы

[1] Byrd William. Relational Programming in miniKanren: Techniques, Applications, and Implementations. — 2009. — 09.

[2] Byrd William E., Holk Eric, Friedman Daniel P. MiniKanren, Live and Untagged: Quine Generation via Relational Interpreters (Programming Pearl) // Proceedings of the 2012 Annual Workshop on Scheme and Functional Programming. — Scheme '12. — New York, NY, USA : Association for Computing Machinery, 2012. — P. 8–29. — URL: https://doi.org/10.1145/2661103.2661105.

[3] Hamilton Geoff. Distillation: Extracting the essence of programs. — 2007. — 01. — P. 61–70.

[4] Jones Neil, Gomard Carsten, Sestoft Peter. Partial Evaluation and Automatic Program Generation. — 1993. — 01. — ISBN: 0-13-020249-5.

[5] Leuschel Michael. Homeomorphic Embedding for Online Termination of Symbolic Methods // The Essence of Computation: Complexity, Analysis, Transformation / Ed. by Torben Æ. Mogensen, David A. Schmidt, I. Hal Sudborough. — Berlin, Heidelberg : Springer Berlin Heidelberg, 2002. — P. 379–403. — ISBN: 978-3-540-36377-4. — URL: https://doi.org/10.1007/3-540-36377-7_17.

[6] Lloyd J.W., Shepherdson J.C. Partial evaluation in logic programming // The Journal of Logic Programming. — 1991. — Vol. 11, no. 3. — P. 217–242. — URL: https://www.sciencedirect.com/science/article/pii/074310669190027M.

[7] Relational Solver for Java Generics Type System / Peter Lozov, Dmitry Kosarev, Dmitry Ivanov, Dmitry Boulytchev // Logic-Based Program Synthesis and Transformation / Ed. by Robert Glück, Bishoksan Kafle. — Cham : Springer Nature Switzerland, 2023. — P. 118–128.

[8] Turchin Valentin F. The Concept of a Supercompiler // ACM Trans. Program. Lang. Syst. — 1986. — jun. — Vol. 8, no. 3. — P. 292–325. — URL: https://doi.org/10.1145/5956.5957.

[9] A Unified Approach to Solving Seven Programming Problems (Functional Pearl) / William E. Byrd, Michael Ballantyne, Gregory Rosenblatt, Matthew Might // Proc. ACM Program. Lang. — 2017. — aug. — Vol. 1, no. ICFP. — 26 p. — URL: https://doi.org/10.1145/3110252.

[10] Verbitskaia Ekaterina, Berezun Daniil, Boulytchev Dmitry. An Empirical Study of Partial Deduction for miniKanren // Electronic Proceedings in Theoretical Computer Science. — 2021. — . — Vol. 341. — P. 73–94. — URL: http://dx.doi.org/10.4204/EPTCS.341.5.