

Анализ времени связывания для реляционных программ

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

Екатерина Вербицкая
JetBrains Research
Санкт-Петербург, 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 програм-

мирования. Это своеобразный шаг к декларативности: достаточно написать одну программу для получения множества целевых функций.

Реляционному программированию родственно логическое, представленное такими языками, как Prolog и Mercury¹ [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.

¹Официальный сайт языка MERCURY: <https://mercurylang.org/>, дата последнего посещения: 15.02.2020

²Официальный сайт языка `miniKANREN`: <http://minikanren.org/>, дата последнего посещения: 15.02.2020

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 различаться 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`.

```
Goal : Goal ∨ Goal
      | Goal ∧ Goal
      | Term ≡ Term
      | invoke Name [Term]
      | fresh [Var] Goal
Term : Var
      | cons Name [Term]
```

Пример программы на языке miniKanren, связывающей три списка, где третий является конкатенацией первых двух, приведен ниже. Мы используем `[]` как сокращение для пустого списка (`cons Nil []`) и `h : t` для конструктора списка с головой `h` и хвостом `t` (`cons Cons [h, t]`), а `[x0, x1, ..., xn]` — для обозначения списка с элементами `x0, x1, ..., xn`.

```
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 `appendo [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 `appendo ? ? [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 `appendo`,

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   $append^o :: [a] \rightarrow [a] \rightarrow [a]$ 
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 $R \subseteq X_0 \times \dots \times X_n$, в котором известны аргументы X_{i_0}, \dots, X_{i_k} , а аргументы X_{j_0}, \dots, X_{j_l} необходимо вычислить, соответствует функция $F : X_{i_0} \rightarrow \dots \rightarrow X_{i_k} \rightarrow [X_{j_0} \times \dots \times X_{j_l}]$, 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:

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

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 Haskell³. Связывание в строке 10 означает, что результат будет вычислен для каждого элемента списка, полученного из рекурсивного вызова функции.

```

1   $append^o :: [a] \rightarrow ([a], [a])$ 
2   $append^o\ x = append_1^o\ x ++ append_2^o\ x$ 
3  where
4     $append_1^o\ y = \mathbf{do}$ 
5      let  $x = []$ 
6      return  $(x, y)$ 
7     $append_1^o\ _ = []$ 
8
9     $append_2^o\ (h : r) = \mathbf{do}$ 
10      $(t, y) \leftarrow append^o\ r$ 
11     let  $x = h : t$ 
12     return  $(x, y)$ 
13    $append_2^o\ _ = []$ 
```

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.

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

```

1  reverso xs sx =
2    (xs ≡ [] ∧ sx ≡ []) ∨
3    (fresh [h, ts, st] (
4      x ≡ h : ts ∧
5      reverso ts st ∧
6      appendo [st, [h], sx]
7    ))

```

Рис. 4. Отношение $revers^o$

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

Цель анализа времени связывания — указать порядок, в котором имена связываются со значениями. Алгоритм принимает на вход программу на miniKanren, данные о входных переменных и каждой переменной ставит в соответствие число. Процесс подбора чисел называется аннотированием.

Если о переменной ничего неизвестно — она аннотируется *Undef*, иначе указывается время связывания — целое положительное число. В начале работы алгоритма известными являются переменными, указанные как входные — они аннотируются 0. В процессе работы алгоритма переменная считается известной (или константой к текущему моменту), если она зависит только от проаннотированных числами переменных — в этом случае она аннотируется числом, большим на 1 максимальной аннотации переменных, от которых она зависит. Таким образом информация о времени связывания распространяется на другие переменные.

На аннотациях выполняется порядок: естественный на положительных числах, *Undef* меньше любой числовой аннотации. Переаннотирование может происходить только в сторону увеличения аннотации: мы можем воспользоваться переменной позже (указав более позднее время связывания), но не можем раньше.

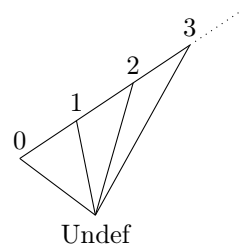


Рис. 5. Полурешётка

А. Алгоритм анализ времени связывания

Реализация разработанного алгоритма доступна на сайте GitHub⁴. Ниже приведено описание алгоритма.

Входные данные алгоритма описаны выше. Выходные — пара из проаннотированной нормализованной (приведённой к дизъюнктивной нормальной форме) цели и стека вызовов — списка аннотированных орпеделений отношений, использованных в цели.

Инициализация алгоритма:

- уникально переименовать все *fresh*-переменные, чтобы избежать перекрытия имён;
- нормализовать для упрощения алгоритма;
- проаннотировать 0 входные переменные

Инициализация цели перед аннотированием:

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

Аннотирование - общий алгоритм:

- создать пустой стек вызовов
- вычисляем аннотации нормализованной цели, пока не достигнем неподвижной точки
- аннотировать нормализованную цель — аннотировать все её дизъюнкты

Аннотировать дизъюнкт — аннотировать все его конъюнкты, при этом аннотации переменных в дизъюнкте должны согласовываться: переменная в конъюнктах одного и того же дизъюнкта должна иметь одну и ту же аннотацию. "Отсортируем" конъюнкты: в начале унификации, а затем вызовы других отношений. Для каждого конъюнкта: согласуем аннотации его переменных с аннотациями всех конъюнктов списка до него, а затем вызовом аннотирование конъюнкта.

Аннотировать конъюнкт — аннотировать либо унификацию, либо вызов отношений.

Аннотирование унификации:

- слева переменная с *Undef*-аннотацией — проаннотировать максимальной аннотацией правого термина, увеличенной на 1

⁴Исходный код алгоритма аннотирования:
github.com/Pluralia/uKanren_translator, дата последнего посещения: 15.02.2020

- слева переменная, аннотированная числом — её значение, увеличенное на 1, установить в качестве значений всех *Undef*-аннотаций правого терма
- слева и справа конструкторы — произвести унификацию их аргументов и запустить рекурсивно аннотирование унификаций
- оставшиеся случаи симметричны

Аннотирование вызова отношения. Для избежания повторного аннотирования информации о проаннотированных в конкретных направлениях отношениях сохраняется в стеке вызовов. Для поиска пары отношение-направление в стеке по имени отношения получим все его направления — среди них попытаемся найти направление, согласованное с нашим. Два направления назовём согласованными, если при сравнении аннотаций их аргументов попарно аннотации одного из направлений будут всегда больше или равны аннотациям другого направления.

Рассмотрим примеры. Пусть есть направления $[x : 0, y : 0, z : \text{Undef}]$ и $[x : 1, y : 0, z : \text{Undef}]$. Они являются согласованными, так как для x $0 < 1$, для y и z аннотации совпадают. Следующие направления несогласованны, так как аннотации x и y больше у первого направления, а аннотация z больше у второго: $[x : 3, y : 2, z : 0]$ и $[x : 0, y : 0, z : 2]$.

Алгоритм аннотирования вызова отношения:

- все термы вызова проаннотированы *Undef* или все проаннотированы числами — вернуть исходную цель
- вызов с таким именем и направлением есть в стеке вызовов — заменить *Undef*-аннотации аргументов вызова на максимальную аннотацию аргументов, увеличенную на 1
- вызова с таким именем и направлением нет в стеке вызовов — добавить его и направление в стек вызовов; проаннотировать цель, вызываемого отношения с учётом обновлённого стека; добавить в стек проаннотированную цель под соответствующим именем и направлением

Существуют отношения, в которых некоторые переменные используются только в отношениях. Пример такого отношения приведён в листинге ???. Укажем y как входную переменную. В этом случае отношения f^o и h^o могут вычисляться в любом порядке независимо друг от друга, однако важно время вычисления g^o . Оно не может вычисляться до вычисления f^o и h^o (неизвестны входные переменные), но может вычисляться между ними или после. При вычислении между ними получим вычисление g^o в прямом или обратном порядке, а после них — g^o играет роль предиката.

При получении такого отношения алгоритм аннотирования может вернуть цель, частично проаннотированную *Undef*. В этом случае необходимо запустить алгоритм ещё раз, поменяв порядок вызова отношений при сортировке конъюнктов в соответствующем

```

1  relo x y z =
2    fo x y ∨
3    ho z y ∨
4    go x z

```

Рис. 6. Пример программы на *miniKANREN*, в которой переменные используются только в отношениях

дизъюнкте. Если при достижении неподвижной точки в цели останутся *Undef*-переменные, цель считается неаннотируемой.

Предложенный алгоритм терминируем, так как благодаря стеку вызовов не происходит повторного аннотирования отношений, а в каждом отношении используется конечное число переменных, конструкторов и вызовов других отношений.

В. Примеры аннотирования

Рассмотрим несколько примеров аннотирования. Числа над переменными обозначают аннотации.

1. Аннотирование цели — вызов append^o в прямом направлении (x и y — входные). Его нет в стеке вызовов, поэтому добавим и запустим рекурсивно аннотирование цели append^o . Проаннотированное append^o в листинге 7. Так как x и y — входные, их аннотации нам известны. Первый дизъюнкт тривиален — рассмотрим второй. На следующем шаге становятся известны аннотации h и t — строка 4. Аннотация h распространяется на ?? строку, а аннотация t — на 6 строку. Здесь же аннотируем r в вызове отношения 6 строки: его направление согласованно с тем, что есть в стеке (см. пример согласованности направлений выше). Распространяем аннотацию r на 5. На последнем шаге аннотируем z из 4. Теперь необходимо вернуть результат аннотирования append^o , основываясь на его дизъюнктах — аннотация каждого аргумента равна максимальной его аннотации среди всех дизъюнктов. В первом дизъюнкте z имеет аннотацию 1, а во втором — 3, поэтому результирующая аннотация равна 3.

```

1  appendo x0 y0 z3 =
2    (x0 ≡ [] ∧ y0 ≡ z1) ∨
3    (fresh [h, t, r] (
4      x0 ≡ h1 : t1 ∧
5      z3 ≡ h1 : r2 ∧
6      appendo [t1, y0, r2]
7    ))

```

Рис. 7. Аннотирование append^o в прямом направлении

2. Проаннотируем ту же цель в обратном направлении (z является входной) в листинге 8. Пусть append^o уже в стеке и z проаннотирована. В первом дизъюнкте x и y имеют аннотацию 1: y определяется входной переменной z , а x зависит от константы. Во втором дизъ-

юнкте при первом шаге становятся известны аннотации h и r — строка 5. Аннотация r доходит до 6 строки — соответствующая пара отношение-направление есть в стеке — аннотируем t и y . Аннотация t распространяется на 4 строку, а на последнем шаге аннотируем x в той же строке.

```

1  appendo x3 y2 z0 =
2  (x1 ≡ [] ∧ y1 ≡ z0) ∨
3  (fresh [h, t, r] (
4    x3 ≡ h1 : t2 ∧
5    z0 ≡ h1 : r1 ∧
6    appendo [t2, y2, r1]
7  ))

```

Рис. 8. Аннотирование $append^o$ в обратном направлении

3. Рассмотрим аннотирование цели — вызов $revers^o$ в обратном направлении. $revers^o$ принимает на вход список и возвращает перевёрнутый список. Его определение приведено в листинге ???. Добавим $revers^o$ по текущему направлению в стек вызовов, вызовем аннотирование и проинициализируем y — как входную переменную. Рассмотрим только второй дизъюнкт. На первом шаге можно попытаться проаннотировать только вызов $append^o$ в строке 6 — известна y . Такого отношения в стеке вызовов нет — добавляем и вызываем аннотирование. Это и есть вызов $append^o$ в обратном направлении, рассмотренный выше. Нам вернутся аннотации r и h — распространяем их по другим конъюнктам. На следующем шаге можем получить аннотацию t из рекурсивного вызова $revers^o$, так как он уже есть в стеке — строка 5. Распространяем аннотацию t и аннотируем x на следующем шаге в строке 4.

```

1  reverso x5 y0 =
2  (x1 ≡ [] ∧ y1 ≡ []) ∨
3  (fresh [h, t, r] (
4    x5 ≡ h2 : t4 ∧
5    reverso [t4, r3] ∧
6    appendo [r3, [h2], y0]
7  ))

```

Рис. 9. Аннотирование $revers^o$ в обратном направлении

С. Обзор существующих решений

Анализ времени связывания — часто применяемая идея анализа программ. Реализация этой идеи зависит от цели применения, а также особенностей анализируемого, что не позволяет использовать существующие решения в нашем случае.

Так, в Mercury [6] анализ времени связывания применяется для специализации программ. Как следствие,

этой реализации не требуется размечать порядок связывания переменных — достаточно разбить их на статические и динамические. Особенностью Mercury являются типы, что так же отразилось на итоговом алгоритме (аннотирование происходит с учётом графа типов, построенного по термам).

Существует анализ времени связывания для лямбда-исчисления [7]. Его цель совпадает с нашей — получить порядок связывания переменных. В алгоритме, предложенном в статье, для этого используются числа от 0 до D — эту идею получилось перенести в анализ времени связывания для miniKanren. Однако, лямбда-исчисление — функциональное программирование — использует совершенно другие конструкции.

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, что влияет на эффективность алгоритма.

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 programming problems (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] P. Thiemann, “A unified framework for binding-time analysis,” in TAPSOFT, 1997.