Санкт-Петербургский государственный университет

Математическое обеспечение и администрирование информационных систем
Кафедра системного программирования

Лозов Петр Алексеевич

Преобразование типизированных функций в реляционную форму

Магистерская диссертация

Научный руководитель:
к. ф.-м. н., доцент Булычев Д. Ю.

Рецензент:
к. ф.-м. н. Павлов В. А.

Санкт-Петербург
2018

SAINT-PETERSBURG STATE UNIVERSITY

Software and Administration of Information Systems
Software Engineering

Petr Lozov

Typed Relation Conversion

Graduation Thesis

Scientific supervisor:
Associate Professor Dmitri Boulytchev

Reviewer:
Ph. D. Vladimir Pavlov

Saint-Petersburg
2018

Оглавление

1. Введение

Реляционное программирование [1] является технологией, основанной на построении программы в виде набора отношений. Реляционная программа может быть исполнена в различных “направлениях”, то есть независимо от того, какие из аргументов отношения известны, а какие необходимо найти. Это делает возможным, например, моделирование вычисления обратных функций.

Хотя многие логические языки программирования, такие как Prolog, Mercury [2] или Curry [3], позволяют использовать некоторые реляционные эффекты, был создан miniKanren [4] — язык, который специально разработан для реляционного программирования. Изначально данный язык являлся небольшим предметно-ориентированным языком (Domain-specific language, DSL) на основе языка Scheme/Racket. Его минимальная реализация [5] содержит менее ста строк кода. Впоследствии miniKanren нашел применение, будучи встроенным во многие языки программирования, среди которых Haskell, Standard ML и OCaml.

Непосредственная разработка реляционных программ является сложной задачей, требующей от разработчика глубокого изучения языка miniKanren и его особенностей. Однако во многих случаях требуемая реляционная программа может быть получена из некоторой функциональной программы автоматически. Таким образом, появляется задача автоматического преобразования функциональных программ в реляционные.

При наличии подобного метода преобразования становится возможным разработка программы на более привычном функциональном языке с последующим преобразованием её в реляционную программу. С одной стороны, данный подход позволяет более гибко использовать функциональные программы за счет их применения в различных направлениях. С другой стороны, этот подход снижает сложность разработки функциональных программ. Действительно, из всех направлений, в которых может исполняться реляционная спецификация, разработчик может выбрать наиболее просто реализуемое направление на функцио-

нальном языке. Например, для реализации генератора всех перестановок элементов заданного списка можно разработать значительно более простой алгоритм сортировки списка, преобразовать его в реляционную форму и исполнить в обратном направлении.

Отметим, что после появления языка miniKanren был создан метод преобразования функциональных программ в реляционные, называемый **Unnesting** [6]. Однако данный метод рассматривает только случай нетипизированных программ и работает для специальных примеров. Более того он никогда не был реализован.

2. Постановка задачи

Целью данной работы является метод реляционного преобразования, который может быть применён к типизированным программам общего вида. Для описания этих программ используется компактный ML-подобный язык (является подмножеством OCaml), оснащенный системой типов Хиндли-Милнера с let-полиморфизмом [7].

Для достижения этой цели были поставлены следующие задачи:

- разработать формальные модели функционального и реляционного языков;
- разработать преобразование типизированных функций в реляционную форму;
- доказать статическую и динамическую корректность преобразования;
- провести апробацию преобразования в реляционную форму.

3. Обзор литературы

3.1. Реляционное программирование и miniKanren

Основной особенностью языка miniKanren [1; 6] является отсутствие различий между аргументами и результатом, что позволяет исполнять программы в различных "направлениях". Такой подход имеет практическое значение: некоторые задачи формулируются гораздо проще, если рассматривать их как запросы к реляционной программе. Существует целый ряд примеров, подтверждающих это наблюдение. К примеру, задача вывода типов для просто-типизированного лямбда-исчисления [8] или задача о выявлении населенности какого-либо типа могут быть сформулированы в виде запросов к более простой в реализации реляционной программе проверки корректности типов. Другим примером может послужить задача генерации "квайнов" [9] — программ, результатом исполнения которых являются они сами. Данная задача может быть представлена как запрос к реляционному интерпретатору, также более простому в сравнении с изначальной задачей. Наконец, генератор всех перестановок элементов данного списка выразим в виде запроса к реляционной сортировке списка.

В контексте данной статьи будет использоваться конкретная реализация языка miniKanren – DSL на основе Objective Caml4, называемый OCanren [10]. Данный язык соответствует оригинальной реализации miniKanren [5]. Также OCanren дополнен конструкцией "Disequality constraint" [11].

Основной синтаксической единицей языка miniKanren является цель (*goal*). Для определения целей существуют несколько конструкций, описания которых представлены ниже.

- Синтаксическая унификация [12] $t_1 \equiv t_2$, где t_1, t_2 — некоторые значения. Унификация выступает в качестве базовой конструкции для построения целей. Если пара значений t_1, t_2 может быть корректно унифирована, то цель считается успешно выполненной. В противном случае цель отвергается.

- Disequality constraint $t_1 \not\equiv t_2$, где t_1, t_2 — некоторые значения. Как и унификация, используется для построения целей; имеет противоположное унификации поведение.

- Дизъюнкция $g_1 \vee g_2$, где g_1, g_2 — некоторые цели. В данном случае обе цели обрабатываются независимо, после чего их решения объединяются.

- Конъюнкция $g_1 \wedge g_2$, где g_1, g_2 — некоторые цели. Этот элемент синтаксиса выполняется последовательно, причем g_2 вычисляется только в случае успешного выполнения цели g_1 и использует результаты её вычисления.

- Конструкция $\underline{fresh}\ (x)\ g$, где x — имя переменной, g — некоторая цель. Данная конструкция используется для ввода переменной, отсутствующей в текущем контексте исполнения (далее подобные переменные будем называть "свежими"), в цель g.

Результат выполнения реляционной программы представляется в виде потока данных, из которого можно запрашивать решения.

```
1 type num = O | S of num
2
3 let rec add a b c =
4   (a ≡ O ∧ b ≡ c) ∨
5   (fresh (a' c')
6      (a ≡ S a') ∧
7      (add a' b c') ∧
8      (c ≡ S c'))
```

Листинг 1: Сложение чисел Пеано

В качестве примера рассмотрим реляционную программу сложения чисел Пеано (листинг 1).

Интерпретировать отношение "`add a b c`" нужно как следующее утверждение: "сумма `a` и `b` равняется `c`". Действительно, в случае, когда `a` равняется нулю, `b` в точности совпадает с `c` (строка 4)

С другой стороны, можно представить a в виде S a' (строка 7) — для этого понадобится "свежая" переменная a' (строка 5). Также понадобится дополнительная переменная c' для обозначения суммы a' и b, что выражается в виде add a' b c' (строка 8). Остается указать, что c должно быть на единицу больше, чем c' (строка 9).

$$\underline{\texttt{fresh}}\ (\texttt{x})\ \texttt{add}\ (\texttt{S}\ \texttt{O})\ (\texttt{S}\ \texttt{O})\ \texttt{x} \Rightarrow [\texttt{x} = \texttt{S}\ (\texttt{S}\ \texttt{O})]$$
(a)

$$\underline{\texttt{fresh}}\ (\texttt{x})\ \texttt{add}\ (\texttt{S}\ (\texttt{S}\ \texttt{O}))\ \texttt{x}\ (\texttt{S}\ (\texttt{S}\ (\texttt{S}\ \texttt{O}))) \Rightarrow [\texttt{x} = \texttt{S}\ \texttt{O}]$$
(b)

$$\underline{\texttt{fresh}}\ (\texttt{x}\ \texttt{y})\ \texttt{add}\ \texttt{x}\ \texttt{y}\ (\texttt{S}\ (\texttt{S}\ \texttt{O})) \Rightarrow \left[\begin{array}{ll} \texttt{x} = \texttt{O}, & \texttt{y} = \texttt{S}\ (\texttt{S}\ \texttt{O}); \\ \texttt{x} = \texttt{S}\ \texttt{O}, & \texttt{y} = \texttt{S}\ \texttt{O}; \\ \texttt{x} = \texttt{S}\ (\texttt{S}\ \texttt{O}), & \texttt{y} = \texttt{O} \end{array}\right]$$
(c)

$$\underline{\texttt{fresh}}\ (\texttt{x})\ \texttt{add}\ (\texttt{S}(\texttt{S}\ (\texttt{S}\ \texttt{O})))\ \texttt{x}\ (\texttt{S}\ (\texttt{S}\ \texttt{O})) \Rightarrow []$$
(d)

Рис. 1: Примеры использования отношения add

Рассмотрим несколько различных целей, построенных с помощью отношения add, и изображенных на рисунке 1. В каждом примере отношение принимает следующие возможные аргументы: константные выражения и "свежие" переменные, внедренные с помощью конструкции fresh. Прежде всего отношение add можно использовать для сложения двух чисел. Для этого в качестве аргументов ему необходимо передать эти числа и "свежую" переменную. В этом случае результатом вычисления цели будет поток, содержащий сумму этих чисел. На рисунке 1a приведен пример суммы двух единиц. Помимо сложения с помощью данного отношения можно произвести вычитание, передав уменьшаемое в качестве третьего аргумента, вычитаемое в качестве первого аргумента и "свежую" переменную, олицетворяющую разность, в качестве второго аргумента. На рисунке 1b приведен пример разности чисел 3 и 2. Также отношение add позволяет сгенерировать все пары слагаемых для фиксированной суммы. Для достижения этой цели передадим

отношению две “свежих” переменных и число. Полученный после вычисления цели поток будет содержать все пары корректных слагаемых. На рисунке 1c приведен пример генерации слагаемых для суммы, равной двум. Наконец, данное отношение позволяет выявить некорректные аргументы, ведь получение в качестве результата вычисления цели пустого потока сигнализирует об отсутствии правильных решений. На рисунке 1d приведен пример попытки прибавить к трем неотрицательное число и получить два.

3.2. Схожие работы

Взаимодействие декларативных языков программирования (а реляционное программирование является декларативным) с языками более прагматичными (например, Java или OCaml) — задача достаточно естественная, так как позволяет совместить выразительность декларативного языка с оптимизированностью и обширной функциональностью императивного или функционального языка. В качестве примера можно упомянуть работы [13; 14], где для описания ограничений на метамодель REAL [15; 16] используется декларативный язык OCL (Object Constraint Language). Данный язык позволил кратко и понятно описать необходимые ограничения, однако оказался непригоден для работы с реальными данными. Поэтому ограничения на языке OCL были преобразованы в скрипты, написанные на императивном языке JavaScript.

В контексте данной работы наиболее показательным декларативным языком программирования является Prolog: во-первых, между ним и языком miniKanren много общего; во-вторых, задача взаимодействия этого языка с другими достаточно исследована. Прежде всего, для языка Prolog существуют методы компиляции в более низкоуровневые языки, например в язык C [17; 18]. Для более высокоуровневого императивного языка Java разработан метод преобразования из Prolog в Java [19], позволяющий увеличить эффективность исполнения декларативной программы. Также формально описан и реализован метод декомпиляции Java Bitecode в Prolog [20]. Помимо этого, проведено

совмещение Prolog и Java посредством встраивания первого языка во второй [21]. Схожее совмещение было проведено для языков Prolog и C# [22].

В случае взаимодействия функционального языка с реляционным можно выделить обратную задачу: преобразование программ в реляционную форму. Решение данной задачи позволит, с одной стороны, использовать привычный язык программирования (в нашем случае OCaml) для описания функций, а, с другой стороны, позволит исполнять программы реляционно (в частности, моделировать вычисление обратных функций). Для данной задачи было предложено решение, называемое **Unnesting** [6].

```
let rec add a b =
  match a with
  | O  →  b
  | S a'  →
     S (add a' b)
```

(a)

```
let rec add a b =
  match a with
  | O  →  b
  | S a'  →
     let q = add a' b in
     S q
```

(b)

```
let rec add a b c =
   (a ≡ O  ∧  b ≡ c)  ∨
   (fresh (a' q)
      (a ≡ S a')  ∧
      (add a' b q)  ∧
      (c ≡ S q))
```

(c)

Рис. 2: Пример исполнения преобразования Unnesting

Данный подход заключается в следующем. Сначала, в программу с помощью конструкции связывания let вводится новая переменная для каждого вложенного подвыражения. Далее, происходит построение реляционной программы: каждая конструкция сопоставления с образцом заменяется на дизъюнкцию, все переменные из шаблона и из let-связывания вводятся с помощью конструкции fresh, каждая функция получает дополнительный аргумент, который унифицируется с результатом. Пример данного преобразования проиллюстрирован

на рисунке 2. Изначальная функциональная программа изображена на рисунке 2a, функциональная программа, полученная после внедрения новых переменных для подвыражений изображена на рисунке 2b, и итоговая реляционная программа изображена на рисунке 2c.

```
let bar y =
  let f x = x in
  let g a = f in
  g A y
```
(a)

```
let bar y r =
  let f x r = x ≡ r in
  let g a r = f ≡ r in
  g A y r
```
(b)

Рис. 3: Некорректный случай для преобразования Unnesting

Однако, не каждая функциональная программа может быть преобразована в реляционную форму с помощью метода Unnesting. Рассмотрим, например, программу, изображённую на рисунке 3a. После применения преобразования будет получена реляционная форма, изображенная на рисунке 3b. Данная форма, очевидно, некорректна, так как она содержит унификацию функции `f` и значения `r`. Для того, чтобы Unnesting построил корректную реляционную программу необходимо провести для тела функции `g` η-расширение. Заметим, что преобразование описанное в секции 5 отлично от Unnesting и использует η-расширение ограниченно (только в одном случае).

4. Входной язык и его реляционное расширение

Предлагаемый нами метод реляционного преобразования основан на идее трансформации программы на функциональном языке в программу на реляционном расширении этого языка. В контексте miniKanren данный подход выглядит вполне естественным, поскольку сам miniKanren, как DSL, использует много важных функций из основного языка (например, абстракцию или конструкторы).

Рассмотрим формальное описание ML-подобного функционального языка, используемого в качестве входного языка для реляционного преобразования. Формальное описание состоит из синтаксиса, правил вывода типов и семантики.

$$
\begin{array}{rcl}
\mathcal{E} & = & x \\
 & & \lambda x.e \\
 & & e_1\ e_2 \\
 & & C^n(e_1, \ldots, e_n) \\
 & & \underline{\texttt{true}} \\
 & & \underline{\texttt{false}} \\
 & & \underline{\texttt{let}}\ x = e_1\ \underline{\texttt{in}}\ e_2 \\
 & & \underline{\texttt{let}}\ \underline{\texttt{rec}}\ x = e_1\ \underline{\texttt{in}}\ e_2 \\
 & & e_1 = e_2 \\
 & & \underline{\texttt{match}}\ e\ \underline{\texttt{with}}\ \{p_i \to e_i\} \\
 & & \\
\mathcal{P} & = & C^n(x_1, \ldots, x_n)
\end{array}
$$

Рис. 4: Синтаксис исходного языка

Синтаксис исходного функционального языка показан на рисунке 4. Он состоит из лямбда-исчисления, обогащенного конструкторами с фиксированной размерностью C^n, двумя предопределенными конструкторами $\underline{\texttt{true}}$ и $\underline{\texttt{false}}$, операцией синтаксического сравнения "=", шаблонами p и конструкциями сопоставления с образцом, а также выражениями для рекурсивных/нерекурсивных let-ссылок.

При использовании сопоставления с образцом доступны только шаблоны вида $C^n(x_1, \ldots, x_n)$. Данное ограничение несущественно, так как шаблоны общего вида выразимы с помощью описанных выше. Также запрещен специальный шаблон wildcard (обозн. "_"), позволяющий игнорировать сопоставляемую ему часть выражения.

Типы:

$$
\begin{array}{lcll}
\mathcal{X} & = & \alpha, \beta, \ldots & \text{(типовые переменные)} \\
\mathcal{D} & = & \texttt{bool}, T^n, \ldots & \text{(конструкторы типов данных)} \\
\mathcal{T} & = & \alpha \mid T^k(t_1, \ldots, t_k) \mid t_1 \to t_2 & \text{(типы)} \\
\mathcal{S} & = & \forall \bar{\alpha}.t & \text{(схемы типов)}
\end{array}
$$

Правила типизации:

$$\Gamma \vdash \underline{\texttt{true}}, \underline{\texttt{false}} : \texttt{bool} \quad [\textsc{Bool}_T] \qquad \frac{\Gamma \vdash e_1 : t \quad \Gamma \vdash e_2 : t}{\Gamma \vdash e_1 = e_2 : \texttt{bool}} \quad [\textsc{Eq}_T]$$

$$\frac{\Gamma \vdash e_i : t_i^C}{\Gamma \vdash C^n(e_1, \ldots, e_n) : t^C} \quad [\textsc{Constr}_T] \qquad \Gamma, x : \forall \bar{\alpha}.t \vdash x : t[\bar{\alpha} \leftarrow \bar{t'}] \quad [\textsc{Var}_T]$$

$$\frac{\Gamma \vdash f : t_1 \to t_2 \quad \Gamma \vdash e : t_1}{\Gamma \vdash f\ e : t_2} \quad [\textsc{App}_T] \qquad \frac{\Gamma,\, x : t_1 \vdash f : t_2}{\Gamma \vdash \lambda x.f : t_1 \to t_2} \quad [\textsc{Abs}_T]$$

$$\frac{\Gamma \vdash e_1 : t_1 \quad \Gamma, x : \forall \bar{\alpha}.t_1 \vdash e_2 : t}{\Gamma \vdash \underline{\texttt{let}}\ x = e_1\ \underline{\texttt{in}}\ e_2 : t},\ \bar{\alpha} = FV(t_1) \setminus FV(\Gamma) \quad [\textsc{Let}_T]$$

$$\frac{\Gamma, f : t_1 \vdash \lambda x.e_1 : t_1 \quad \Gamma, f : \forall \bar{\alpha}.t_1 \vdash e_2 : t}{\Gamma \vdash \underline{\texttt{let}}\ \underline{\texttt{rec}}\ f = \lambda x.e_1\ \underline{\texttt{in}}\ e_2 : t},\ \bar{\alpha} = FV(t_1) \setminus FV(\Gamma) \quad [\textsc{LetRec}_T]$$

$$\frac{\Gamma \vdash e : t^C \quad \Gamma, x_1^i : t_1^{C_i}, \ldots, x_{k_i}^i : t_{k_i}^{C_i} \vdash e_i : t}{\Gamma \vdash \underline{\texttt{match}}\ e\ \underline{\texttt{with}}\ \{C_i^{k_i}(x_1^i, \ldots, x_{k_i}^i) \to e_i\} : t} \quad [\textsc{Match}_T]$$

Рис. 5: Правила типизации для входного языка

Данный язык оснащен системой вывода типов Хиндли-Милнера, правила которой описаны на рисунке 5. Данная система вывода типов поддерживает типовые переменные, функциональные типы, а также набор неявно определенных алгебраических типов данных T^k, причем каждый конструктор C^n принадлежит ровно одному типу, и конструкторы $\underline{\texttt{true}}$ и $\underline{\texttt{false}}$ принадлежат выделенному алгебраическому типу `bool`.

В правиле $\textsc{Constr}_T$ предполагается, что тип t^C представим в виде $T^k(t_1, \ldots, t_k)$, где каждый из типов t_i восстанавливается из типов t_i^C аргументов конструктора C^n.

Значения:

$$\mathcal{V} = C^n(v_1, \ldots, v_n) \mid \lambda x.e \mid \mu f \lambda x.e \mid \underline{\texttt{true}} \mid \underline{\texttt{false}}$$

Контексты:

$$\mathcal{C} = \square\ e \mid v\ \square \mid \underline{\texttt{let}}\ x = \square\ \underline{\texttt{in}}\ e \mid \underline{\texttt{match}}\ \square\ \underline{\texttt{with}}\ \{p_i \to e_i\} \mid C^n(\bar{v}, \square, \bar{e}) \mid \square = e \mid v = \square$$

Стек контекстов:

$$\mathcal{S} = \epsilon \mid \mathcal{C} : \mathcal{S}$$

Состояния:

$\langle \mathcal{S}, e \rangle$ (стек контекстов, выражение); $\langle \epsilon, e \rangle$ (начальное состояние); $\langle \epsilon, v \rangle$ (финальное состояние)

Переходы:

$$\langle C : \mathcal{S},\ v \rangle \to \langle \mathcal{S},\ C[v] \rangle \quad [\textsc{Value}]$$

$$\langle \mathcal{S},\ f\ e \rangle \to \langle \square\ e : \mathcal{S},\ f \rangle \quad [\textsc{AppL}]$$

$$\langle \mathcal{S},\ v\ e_2 \rangle \to \langle v\ \square : \mathcal{S},\ e_2 \rangle \quad [\textsc{AppR}]$$

$$\langle \mathcal{S},\ e_1 = e_2 \rangle \to \langle \square = e_2 : \mathcal{S},\ e_1 \rangle \quad [\textsc{EqL}]$$

$$\langle \mathcal{S},\ v = e \rangle \to \langle v = \square : \mathcal{S},\ e \rangle \quad [\textsc{EqR}]$$

$$\langle \mathcal{S},\ v = v \rangle \to \langle \mathcal{S},\ \underline{\texttt{true}} \rangle \quad [\textsc{EqTrue}]$$

$$\langle \mathcal{S},\ v_1 = v_2 \rangle \to \langle \mathcal{S},\ \underline{\texttt{false}} \rangle,\ v_1 \neq v_2 \quad [\textsc{EqFalse}]$$

$$\langle \mathcal{S},\ (\lambda x.e)\ v \rangle \to \langle \mathcal{S},\ e[x \leftarrow v] \rangle \quad [\textsc{Beta}]$$

$$\langle \mathcal{S},\ (\mu f \lambda x.e)\ v \rangle \to \langle \mathcal{S},\ e[f \leftarrow \mu f \lambda x.e,\ x \leftarrow v] \rangle \quad [\textsc{Mu}]$$

$$\langle \mathcal{S},\ C^n(v_1, \ldots, v_{k-1}, e_k, \ldots, e_n) \rangle \to \langle C^n(v_1, \ldots, v_{k-1}, \square, \ldots, e_n) : \mathcal{S},\ e_k \rangle \quad [\textsc{Constr}]$$

$$\langle \mathcal{S},\ \underline{\texttt{let}}\ x = e_1\ \underline{\texttt{in}}\ e_2 \rangle \to \langle \underline{\texttt{let}}\ x = \square\ \underline{\texttt{in}}\ e_2 : \mathcal{S},\ e_1 \rangle \quad [\textsc{Let}]$$

$$\langle \mathcal{S},\ \underline{\texttt{let}}\ x = v\ \underline{\texttt{in}}\ e \rangle \to \langle \mathcal{S},\ e[x \leftarrow v] \rangle \quad [\textsc{LetVal}]$$

$$\langle \mathcal{S},\ \underline{\texttt{let}}\ \underline{\texttt{rec}}\ f = \lambda x.e_1\ \underline{\texttt{in}}\ e_2 \rangle \to \langle \mathcal{S},\ e_2[f \leftarrow \mu f \lambda x.e_1] \rangle \quad [\textsc{LetRec}]$$

$$\langle \mathcal{S},\ \underline{\texttt{match}}\ e\ \underline{\texttt{with}}\ \{p_i \to e_i\} \rangle \to \langle \underline{\texttt{match}}\ \square\ \underline{\texttt{with}}\ \{p_i \to e_i\} : \mathcal{S},\ e \rangle \quad [\textsc{Match}]$$

$$\left\langle \mathcal{S},\ \underline{\texttt{match}}\ C_k^{n_k}(v_1, \ldots, v_{n_k})\ \underline{\texttt{with}}\ \{C_i^{n_i}(x_1^i, \ldots, x_{n_i}^i) \to e_i\} \right\rangle \to \left\langle \mathcal{S},\ e_k[x_j^k \leftarrow v_j] \right\rangle \quad [\textsc{MatchVal}]$$

Рис. 6: Семантика входного языка

Также в правиле Матсн$_T$ типы каждого образца $C_i^{k_i}(x_1^i, \ldots, x_{k_i}^i)$ должны быть равны типу t^C, а $t_j^{C_i}$ являются типом j-ого аргумента конструктора C_i. Правило Eq$_T$ требует одинакового типа для обоих аргументов.

Семантика данного языка, представленная в нотации Матиаса Феллейсена [23] (рисунок 6), является системой переходов между состояниями. Отношение перехода $\langle S, e\rangle \to \langle S', e'\rangle$ описывает один шаг вычисления выражения e в стеке контекстов S, после которого будет получено новое выражение e' с обновленным стеком контекстов S'. Контекст представляет из себя выражение с уникальной дырой; неформально говоря, стек контестов описывает путь вычисления выражения от внешнего уровня до места, где в текущий момент остановлено вычисление. Для контекста C и выражения e обозначим $C[e]$ — полное выражение без дыр, полученное путем подстановки e вместо уникальной дыры в C. Для состояния $\langle C_1 : \ldots : C_n, e\rangle$ полным выражением является $C_n[\ldots[C_1[e]]\ldots]$, которое является промежуточным результатом вычисления.

В данной семантике правила Beta, Mu, LetVal, LetRec и MatchVal используют подстановку, которая согласуется с переменными в лямбда-абстракции и `let`-конструкции. В правиле MatchVal предполагается, что каждый конструктор-образец уникален — это значимое отличие от стандартной семантики сопоставления с образцом, где шаблоны рассматриваются сверху вниз до первого успешного сопоставления.

Наконец, выражение e вычисляется к результирующему значению v если $\langle \epsilon, e\rangle \to^* \langle \epsilon, v\rangle$, где ϵ — пустой стек, "$\to^*$" — рефлексивно-транзитивное замыкание отношения "$\to$".

Реляционное расширение добавляет пять стандартных конструкций языка miniKanren для построения целей, синтаксис которых отображен на рисунке 7.

$$
\begin{aligned}
\mathcal{E} \mathrel{+}= \ & \texttt{fresh}\ (x)\ e \\
& e_1 \equiv e_2 \\
& e_1 \not\equiv e_2 \\
& e_1 \vee e_2 \\
& e_1 \wedge e_2
\end{aligned}
$$

Рис. 7: Синтаксис реляционного расширения

Вследствие добавления конструкций miniKantren к конструкциям функционального языка, становится возможным построение всевозможных смешанных выражений, к примеру, $(\lambda \mathtt{x.x} \wedge \lambda \mathtt{y.y})$. Для устранения подобных некорректных выражений была расширена система типов для исходного языка, что описано на рисунке 8. Фактический, данный подход следует реализации языка OCaml, где строгая система типов позволяет исключить большинство некорректных программ во время компиляции. Также система типов была дополнена специальным типом $\mathfrak{G}$, олицетворяющим результат вычисления отношения.

Типы:

$$\mathcal{T} \ += \ \mathfrak{G}$$

Правила типизации:

$$\frac{\Gamma, x : l \vdash e : \mathfrak{G}}{\Gamma \vdash \underline{\mathtt{fresh}}\ (x)\ e : \mathfrak{G}} \quad [\textsc{Fresh}_T]$$

$$\frac{\Gamma \vdash e_1 : l \quad \Gamma \vdash e_2 : l}{\Gamma \vdash e_1 \equiv e_2 : \mathfrak{G}} \quad [\textsc{Unify}_T] \qquad \frac{\Gamma \vdash e_1 : l \quad \Gamma \vdash e_2 : l}{\Gamma \vdash e_1 \not\equiv e_2 : \mathfrak{G}} \quad [\textsc{Disequality}_T]$$

$$\frac{\Gamma \vdash e_1 : \mathfrak{G} \quad \Gamma \vdash e_2 : \mathfrak{G}}{\Gamma \vdash e_1 \wedge e_2 : \mathfrak{G}} \quad [\textsc{Conjunction}_T] \qquad \frac{\Gamma \vdash e_1 : \mathfrak{G} \quad \Gamma \vdash e_2 : \mathfrak{G}}{\Gamma \vdash e_1 \vee e_2 : \mathfrak{G}} \quad [\textsc{Disjunction}_T]$$

Рис. 8: Правила типизации для реляционного расширения

Семантика расширенного языка представлена на рисунке 9. Прежде всего, было расширено состояние: помимо стека контекстов и текущего выражения состояние теперь содержит множество использованных *семантических переменных* Σ и *реляционное состояние* σ. Семантические переменные вводятся и заменяют синтаксические переменные после каждого исполнения конструкции $\underline{\mathtt{fresh}}$. Также расширенная семантика в отличие от исходной является недетерминированной, так как к состаянию вида $\langle \Sigma, \mathcal{S}, e_1 \vee e_2, \sigma \rangle$ применимо два правилаDisjL и DisjR. Аналогично, к состаянию вида $\langle \Sigma, \mathcal{S}, e_1 \wedge e_2, \sigma \rangle$ могут быть применены правила ConjStartL и ConjStartR. Таким образом результатом вычисления реляционной программы может иметь пустое, конечное или бесконечное множество успешно вычисленных состояний.

Семантические переменные:

$$\mathfrak{S} = \mathfrak{s}_1, \mathfrak{s}_2, \ldots$$

$\Sigma, \Sigma' \cdots \subset 2^{\mathcal{S}}$ (множества выделенных семантических переменных)

$\langle \Sigma', \mathfrak{s} \rangle \leftarrow \underline{\texttt{new}}\ \Sigma,\ \Sigma' = \Sigma \cup \{\mathfrak{s}\},\ \mathfrak{s} \notin \Sigma$ (выделение новой семантической переменной)

Значения:

$$\mathcal{V} \mathrel{+}= \underline{\texttt{success}} \mid \mathfrak{s}$$

Контексты:

$$\mathcal{C} \mathrel{+}= \square \equiv e \mid v \equiv \square \mid \square \not\equiv e \mid v \not\equiv \square \mid \square \wedge e \mid e \wedge \square$$

Состояния:

$\langle \Sigma, \mathcal{S}, e, \sigma \rangle$ (мн-во семантических переменных, стек контекстов, выражение, логическое состояние)

$\langle \emptyset, \epsilon, e, \iota \rangle$ (начальное состояние)

Переходы:

$$\langle \Sigma,\ \mathcal{S},\ \underline{\texttt{fresh}}\ (x)\ e,\ \sigma \rangle \rightsquigarrow \langle \Sigma',\ \mathcal{S},\ e[x \leftarrow \mathfrak{s}],\ \sigma \rangle,\ \langle \Sigma', \mathfrak{s} \rangle \leftarrow \underline{\texttt{new}}\ \Sigma \qquad [\textsc{Fresh}]$$

$$\langle \Sigma,\ \mathcal{S},\ e_1 \equiv e_2,\ \sigma \rangle \rightsquigarrow \langle \Sigma,\ \square \equiv e_2 : \mathcal{S},\ e_1,\ \sigma \rangle \qquad [\textsc{UnifyL}]$$

$$\langle \Sigma,\ \mathcal{S},\ v \equiv e,\ \sigma \rangle \rightsquigarrow \langle \Sigma,\ v \equiv \square : \mathcal{S},\ e,\ \sigma \rangle \qquad [\textsc{UnifyR}]$$

$$\langle \Sigma,\ \mathcal{S},\ v_1 \equiv v_2,\ \sigma \rangle \rightsquigarrow \langle \Sigma,\ \mathcal{S},\ \underline{\texttt{success}},\ \sigma' \rangle,\ \mathbf{unify}\,(\sigma,\ v_1,\ v_2) = \sigma' \qquad [\textsc{Unify}]$$

$$\langle \Sigma,\ \mathcal{S},\ e_1 \not\equiv e_2,\ \sigma \rangle \rightsquigarrow \langle \Sigma,\ \square \not\equiv e_2 : \mathcal{S},\ e_1,\ \sigma \rangle \qquad [\textsc{DisEqL}]$$

$$\langle \Sigma,\ \mathcal{S},\ v \not\equiv e,\ \sigma \rangle \rightsquigarrow \langle \Sigma,\ v \not\equiv \square : \mathcal{S},\ e,\ \sigma \rangle \qquad [\textsc{DisEqR}]$$

$$\langle \Sigma,\ \mathcal{S},\ v_1 \not\equiv v_2,\ \sigma \rangle \rightsquigarrow \langle \Sigma,\ \mathcal{S},\ \underline{\texttt{success}},\ \sigma' \rangle,\ \mathbf{diseq}\,(\sigma,\ v_1,\ v_2) = \sigma' \qquad [\textsc{DisEq}]$$

$$\langle \Sigma,\ \mathcal{S},\ e_1 \vee e_2,\ \sigma \rangle \rightsquigarrow \langle \Sigma,\ \mathcal{S},\ e_1,\ \sigma \rangle \qquad [\textsc{DisjL}]$$

$$\langle \Sigma,\ \mathcal{S},\ e_1 \vee e_2,\ \sigma \rangle \rightsquigarrow \langle \Sigma,\ \mathcal{S},\ e_2,\ \sigma \rangle \qquad [\textsc{DisjR}]$$

$$\langle \Sigma,\ \mathcal{S},\ e_1 \wedge e_2,\ \sigma \rangle \rightsquigarrow \langle \Sigma,\ \square \wedge e_2 : \mathcal{S},\ e_1,\ \sigma \rangle \qquad [\textsc{ConjStartL}]$$

$$\langle \Sigma,\ \mathcal{S},\ e_1 \wedge e_2,\ \sigma \rangle \rightsquigarrow \langle \Sigma,\ e_1 \wedge \square : \mathcal{S},\ e_2,\ \sigma \rangle \qquad [\textsc{ConjStartR}]$$

$$\langle \Sigma,\ \mathcal{S},\ \underline{\texttt{success}} \wedge e,\ \sigma \rangle \rightsquigarrow \langle \Sigma,\ \mathcal{S},\ e,\ \sigma \rangle \qquad [\textsc{ConjL}]$$

$$\langle \Sigma,\ \mathcal{S},\ e \wedge \underline{\texttt{success}},\ \sigma \rangle \rightsquigarrow \langle \Sigma,\ \mathcal{S},\ e,\ \sigma \rangle \qquad [\textsc{ConjR}]$$

Рис. 9: Семантика для реляционного расширения

Реляционное состояние используется при исполнении унификации и desequality constraint. Оно состоит из положительной подстановки и множества отрицательных подстановок. Положительная подстановка сопоставляет семантическим переменным выражения, с которыми они связаны. Каждая из отрицательных также сопоставляет семантические переменные и выражения и является ограничением для всех последующих исполнений унификаций.

Как видно из описания семантики (рисунок 9) , реляционное состояние обновляется только с помощью функций **unify** (правило [Unify]) и **diseq** (правило [DisEq]). Обе эти функции обобщают алгоритм синтаксической унификации [12]. Ниже представлено неформальное описание работы данных функций.

Функция **unify** унифицирует свои аргументы в контексте текущей положительной подстановки, создавая новую положительную подстановку. В случае успешной унификации эта функция сравнивает новую подстановку со всеми отрицательными подстановками. Если унификация прошла успешно и ни одна отрицательная подстановка не вложена в новую положительную, то функция **unify** успешно выполняется и возвращает обновленное реляционное состояние с новой положительной подстановкой. В противном случае, **unify** не возвращает ничего, и правило [Unify] не может быть применено, следовательно, вычисление для текущего состояния не может быть продолжено. Функция **diseq** также унифицирует свои аргументы в контексте текущей положительной постановке, в результате чего возникает новая отрицательная подстановка. Если эта отрицательная подстановка отличается от положительной подстановки, то **diseq** завершается и возвращает новое реляционное состояние, пополненное отрицательной подстановкой. В противном случае, правило [DisEq] не может быть применено, и вычисление для текущего состояния не может быть продолжено. Подробнее эти функции описаны в работе [11].

Отметим, что все существующие правила семантики исходного языка дополняются множеством семантических переменных и реляционным состоянием, но не используют их.

Наконец, для замкнутой реляционной программы g типа $\mathfrak{G}$ и реляционного состояния сигма σ, определим $g \rightsquigarrow^r \sigma$ верным тогда и только тогда, когда

$$\langle \emptyset, \epsilon, g, \iota \rangle \rightsquigarrow^* \langle \Sigma, \epsilon, \underline{\texttt{success}}, \sigma \rangle \text{ для некоторого } \Sigma$$

где "$\rightsquigarrow^*$" рефлексивно-транзитивное замыкание операции "$\rightsquigarrow$".

5. Преобразование функциональных программ в реляционную форму

Прежде чем описать преобразование функциональных программ в реляционные, сформулируем несколько ограничений для входных программ. Функциональные программы, как правило, оперируют значениями высшего порядка, в то время как miniKanren ограничен унификацией первого порядка. Поэтому не всякая функциональная программа может быть преобразована в реляционную форму.

Неформально говоря, необходимо исключить значения, которые содержат в своей структуре значения высшего порядка. Это выражается в виде следующих ограничений на преобразовываемую программу:

- тип любого конструктора должен содержать либо типовые переменные, либо типовые константы;
- конструкторы и полиморфная операция сравнения могут быть применены только к значениям первого порядка;
- все `match`-выражения должны быть первого порядка.

Первые два ограничения сужают полиморфизм для реляционных программ: все типовые переменные могут быть заменены только на типы выражений первого порядка (это ограничение, конечно, достаточно, но не необходимо).

Третье ограничение несущественно и введено только для упрощения. Если `match`-выражение имеет тип высшего порядка, то его всегда можно преобразовать, используя η-расширение:

$$\texttt{match}\ e\ \texttt{with}\ \{p_i\ \ \to e_i\} \rightsquigarrow \lambda\bar{x}.\texttt{match}\ e\ \texttt{with}\ \{p_i\ \ \to e_i\,\bar{x}\},$$

где $\bar{x}$ — это вектор новых переменных, отсутствующих в выражениях e, e_i, and p_i. Отметим, что реализация, описанная в разделе 6, исполняет это расширение для `match`-выражений, если оно является выражением высшего порядка. Это единственный случай, когда для преобразования используется тип функциональной программы и η-расширение.

Основная идея преобразования может быть проиллюстрирована на уровне типов: выражение типа t в исходном языке будет преобразовано в выражение типа $[\![t]\!]^t$ в реляционном расширении языка, где преобразование $[\![\bullet]\!]^t$ определяется следующим образом:

$$\begin{array}{rcl} [\![g]\!]^t & = & g \to \mathfrak{G} \\ [\![t_1 \to t_2]\!]^t & = & [\![t_1]\!]^t \to [\![t_2]\!]^t \end{array}$$

Другими словами, выражение первого порядка будет преобразовано в одноместную функцию, возвращающую значения типа $\mathfrak{G}$. Неформальная семантика данной функции состоит в том, чтобы сопоставить аргументу исходное значение. Например, константа `Nil` будет преобразована в функцию $\lambda\ q\ .\ q \equiv \texttt{Nil}$.

Теперь рассмотрим преобразование выражений, обозначаемое $[\![\bullet]\!]^c$.

$$\begin{array}{rcl} [\![x]\!]^c & = & x \\ [\![\lambda x.e]\!]^c & = & \lambda x.[\![e]\!]^c \\ [\![f\ e]\!]^c & = & [\![f]\!]^c\ [\![e]\!]^c \\ [\![\underline{\texttt{let}}\ x = e_1\ \underline{\texttt{in}}\ e_2]\!]^c & = & \underline{\texttt{let}}\ x = [\![e_1]\!]^c\ \underline{\texttt{in}}\ [\![e_2]\!]^c \\ [\![\underline{\texttt{let}}\ \underline{\texttt{rec}}\ f = \lambda x.e_1\ \underline{\texttt{in}}\ e_2]\!]^c & = & \underline{\texttt{let}}\ \underline{\texttt{rec}}\ f = [\![\lambda x.e_1]\!]^c\ \underline{\texttt{in}}\ [\![e_2]\!]^c \end{array}$$

Первые пять правил не меняют структуру выражения, применяя преобразование к подвыражаениям.

$$\begin{array}{rcll} [\![C^k(e_1, \ldots, e_k)]\!]^c & = & \lambda\ q.\underline{\texttt{fresh}}\ (q_1 \ldots q_k) & ([\![e_1]\!]^c\ q_1)\ \wedge \\ & & & \ldots \\ & & & ([\![e_k]\!]^c\ q_k)\ \wedge \\ & & & (q \equiv C^n(q_1, \ldots, q_k)) \end{array}$$

В случае конструктора, все выражения e_i являются выражениями первого порядка. Следовательно, их реляционные образы будут одноместными функциями, возвращающими цель. Для вычисления этих значений необходимо создать набор "свежих" переменных (по одной, для каждого выражения) и передать их образам в качестве аргументов. Все образы с переменными соединяем оператором конъюнкции. Результатом преобразования всего конструктора также должна быть одноместная функция, возвращающая цель, поэтому окружаем абстракцией по переменной q полученное выше выражение, а также связываем

переменную q с конструктором, примененным к созданным ранее переменным.

$$\left[\!\left[\underline{\texttt{match}}\ e\ \underline{\texttt{with}} \atop \{C_i^{n_i}(x_1^i, \ldots, x_{n_i}^i)\ \rightarrow\ e_i\} \right]\!\right]^c = \begin{array}{l} \lambda\ q.\underline{\texttt{fresh}}\ (q_e) \\ \quad (\llbracket e \rrbracket^c\ q_e)\ \wedge \\ \quad \bigvee_i\ ((\underline{\texttt{fresh}}\ (q_1^i \ldots q_{n_i}^i) \\ \qquad\qquad (q_e \equiv C_i^{n_i}(q_1^i, \ldots, q_{n_i}^i))\ \wedge \\ \qquad\qquad (\lambda\ x_1^i \ldots x_{n_i}^i . \llbracket e_i \rrbracket^c) \\ \qquad\qquad\quad (\equiv q_1^i)\ \ \ldots\ \ (\equiv q_{n_i}^i)\ \ q \\ \qquad)) \end{array}$$

Правило для преобразования сопоставления с образцом работает аналогичным образом. Во-первых, скрутини (разбираемое значение e) должно быть выражением первого порядка (так как оно сопоставляется конструкторам). Создадим "свежую" переменную q_e и свяжем её со значением скрутини так же, как и в предыдущем случае. Далее, для каждой ветки создадим несколько "свежих" переменных q_j^i (по одной для каждой переменной в образце данной ветки) и выразим сопоставление образца с помощью оператора дизъюнкции, используя эти переменные и соответствующий конструктор. Наконец, тело ветки e_i — это выражение со свободными переменными, соответствующими тем, что указаны в образце. Поэтому преобразуем выражение e_i и и окружим результат абстракциями, замыкающими все эти переменные и получим функцию. Теперь необходимо связать q_j^i со свободными переменными из образца. Для этого применим описанную выше функцию к функциям, возвращающим цель ($\equiv q_j^i$). В конечном итоге получим функцию, возвращающую цель, которую применим ко внешней переменной q, олицетворяющей результат исходного сопоставления с образцом.

$$\llbracket e_1 = e_2 \rrbracket^c \ = \ \lambda\ q.\underline{\texttt{fresh}}\ (q_1\ q_2) \quad \begin{array}{l} \llbracket e_1 \rrbracket^c\ q_1\ \ \wedge \\ \llbracket e_2 \rrbracket^c\ q_2\ \ \wedge \\ ((q_1 \equiv q_2\ \ \wedge q \equiv \underline{\texttt{true}})\ \ \vee \\ \ (q_1 \not\equiv q_2\ \ \wedge q \equiv \underline{\texttt{false}}) \\ ) \end{array}$$

Последнее правило следует тому же шаблону: оба аргумента полиморфного сравнения преобразуются в функции, возвращающие цель, причем их аргументы будут иметь одинаковый тип выражения первого порядка. Применим эти функции к “свежим” переменным и выполним разбор двух случаев: сравниваемые выражения равны, либо не равны. Отметим, что это единственный случай использования конструкции disequality constraint.

Интересным свойством данного преобразования в реляционную форму является сохранение выражения неизменным в том случае, когда оно не содержит конструкторов, сравнения и сопоставления с образцом. Таким образом, множество полезных функций высшего порядка — применение, композиция, неподвижная точка — уже являются реляционными и могут быть использованы в реляционных спецификациях.

Другое свойство состоит в том, что это преобразование в реляционную форму является композиционной (действительно, реляционный образ применения есть применение реляционных образов). Это означает, что реляционное преобразование совместимо с раздельной компиляцией — несколько исходных файлов могут быть преобразованы независимо, не теряя возможности работать должным образом при их объединении.

Также, интересным является тот факт, что результат преобразования в реляционную форму исполняется детерминировано в прямом направлении. Таким образом преобразование в реляционную форму вызывает константное замедление при прямом исполнении.

6. Доказательство статической и динамической корректности преобразования функций реляционную форму

Первая теорема, которая доказана в данном разделе, подтверждает корректность типизации преобразованной программы при условии корректности типизации исходной функциональной программы.

Теорема 1 (о статической корректности)**.** Если выражение e имеет тип t в исходном языке, тогда $[\![e]\!]^c$ имеет тип $[\![t]\!]^t$ в реляционном расширении.

Прежде чем приступить к доказательству теоремы сформулируем определение и лемму.

Определение 1. Пусть Γ - контекст, возникающий при выводе типа функциональной программы P в узле X синтаксического дерева программы. Тогда $[\![\Gamma]\!] = \{(x : [\![t]\!]^t) \mid (x : t) \in \Gamma\}$ — реляционный образ контекста Γ.

Лемма 1. Пусть при выводе типа функциональной программы P в узле X синтаксического дерева программы имеем состояние $\Gamma \vdash e : t$. Также при выводе типа реляционной программы $[\![P]\!]^c$ в узле $[\![X]\!]^c$, являющемся образом узла X, имеем состояние $\bar{\Gamma} \vdash \bar{e} : \bar{t}$. Тогда $[\![\Gamma]\!] \subset \bar{\Gamma}$.

Доказано индукцией по длине пути от корня синтаксического дерева программы P до узла X.

Данная лемма подтвержает тот факт, что при выводе типа реляционного образа будут типизированы все переменные из исходной программы корректными типами.

Доказательство самой теоремы было преведено с помощью структурной индукции с использованием леммы 1 в базе индукции.

Теорема 2 (о частичной динамической корректности)**.** Если выражение первого порядка e имеет тип t, а также существует значение первого порядка v такое, что $e \rightsquigarrow^f v$, тогда

$\underline{\texttt{fresh}}\ (x)\ (\llbracket e \rrbracket^c\ x) \rightsquigarrow^r (\theta, \emptyset)$, и $\theta(\mathfrak{s}) = v$, где $\mathfrak{s}$ – семантическая переменная, ассоциированная с x на первом шаге вычисления.

Прежде всего прокомментируем тот факт, что множество отрицательных подстановок является пустым. Пополнение данного множества возможно только при выполнении конструкции disequality constraint. Эта конструкция может быть исполнена в преобразованной программе, только если исходная программа содержит синтаксическое сравнение, примененное к своим аргументам. При исполнении преобразованной реляционной программы в прямом направлении (последний аргумент является свежей переменной, остальные аргументы полностью определены) при выполнении конструкции desequality constraint оба аргумента являются замкнутыми, что приводит к немедленному разрешению desequality constraint без пополнения изначально пустого множества отрицательных подстановок.

Также отметим, что данную теорему нельзя доказать индукцией по длине вывода, ведь в случае, например, применения его левое подвыражение не является выражением первого порядка. Это ограничение можно было бы снять, если бы можно было доказать следующее обобщение:

$$p \rightsquigarrow^f f \Rightarrow \llbracket p \rrbracket^c \rightsquigarrow^r \llbracket f \rrbracket^c$$

для произвольного p любого типа. Это утверждение, однако, оказалось ложным — выражение $\texttt{C}\ ((\lambda\ \texttt{x.x})\ \texttt{A})$ можно предъявить в качестве примера.

Причина данной проблемы заключается в том, что при преобразовании происходит *функционализация* конструкторов, сопоставления с образцом и синтаксического сравнения и, следовательно, изменяется порядок вычисления реляционного образа в сравнении с исходной функциональной программой. Таким образом при доказательстве необходимо решить эту проблему.

Во-первых, была разработана модифицированная семантика функционального языка, порядок вычисления в которой приближен к по-

рядку вычисления реляционного образа. Данная семантика была названа *откладывающей*, она откладывает вычисление конструкторов, сопоставления с образом и синтаксического сравнения. Эта семантика может быть получена из исходной функциональной семантики в два шага. Сначала необходимо рассмотреть сокращенную версию оригинальной функциональной семантики, которая обрабатывает конструкторы, сопоставления с образцом и синтаксические сравнения как вычисленные значения. Затем отложенная семантика — это итеративное применение сокращенной версии к аргументам этих новых значений (аргументы конструкторов или синтаксического сравнения, а также сопоставляемое значение сопоставления с образцом).

Далее, отметим, что если выражение первого порядка вычисляется в некоторое значение в исходной семантике, то он также вычисляется к тому же значению в откладывающей семантике. Это свойство основано на следующих наблюдениях:

- свойства **progress** и **type preservation** [7] для обеих семантик (могут быть доказаны стандартным способом);
- свойство Черча-Россера [7; 24] для лямбда-исчисления;
- тот факт, что откладывающая семантика применяет подмножество правил исходной семантики.

Теперь приступим к доказательству теоремы с помощью симуляции между исходной программой в откладывающей семантике и реляционного образа в реляционной семантики. Перед этим сформулируем несколько лемм и определений.

Лемма 2. Разделим все контексты на два дизъюнктных множества — функциональные (1) и атомарные (2).

$$C_f = \square\ e \mid v\ \square \mid \underline{\texttt{let}}\ x = \square\ \underline{\texttt{in}}\ e \tag{1}$$

$$C_g = \underline{\texttt{match}}\ \square\ \underline{\texttt{with}}\ \{p_i \to e_i\} \mid C^n(\bar{v}, \square, \bar{e}) \mid \square = \texttt{e} \mid \texttt{v} = \square \tag{2}$$

Пусть $\langle \mathcal{S}, e \rangle$ — произвольное состояние последовательности вычислений в откладывающей семантике. Тогда $\mathcal{S} = C_f^* C_g^*$.

Другими словами, в процессе вычисления в откладывающей семантики стек контекстов можно разделить на два (возможно, пустых) сегмента: все атомарные контексты расположены ниже всех функциональных. Доказано индукцией по длине вывода.

Определение 2. Разделим все выражения исходного языка на два дизъюнктных множества — функциональные (3) и атомарные (4).

$$e_1\, e_2 \mid \lambda x.e \mid \mu f.\lambda x.e \mid \underline{\texttt{let}}\ x = e_1\ \underline{\texttt{in}}\ e_2 \mid \underline{\texttt{let}}\ \underline{\texttt{rec}}\ f = \lambda x.e_1\ \underline{\texttt{in}}\ e_2 \qquad (3)$$

$$e_1 = e_2 \mid \underline{\texttt{match}}\ e\ \underline{\texttt{with}}\ \{p_i \to e_i\ \} \mid \mathtt{C}^k\ (e_1 \ldots e_k) \qquad (4)$$

Определение 3. Расширенное преобразование выражение в подстановке $[\![\bullet]\!]_\theta$ определяется следующим образом:

$$\begin{array}{lcl} [\![p]\!]_\theta & = & [\![p]\!]^c \\ [\![v]\!]_\theta & = & (\lambda x.x \equiv \mathfrak{s}),\ \text{if}\ \ \theta(\mathfrak{s}) = v \end{array}$$

Здесь θ — подстановка, p — произвольное функциональное выражение, v — произвольное значение первого порядка в исходной семантике (т. е. состав конструкторов). Заметим, что случаи в этом определении не дизъюнктны, а во втором случае может быть более одной переменной с запрошенным свойством, поэтому расширенное преобразование определяет набор реляционных выражений.

Лемма 3. Пусть f, e — два произвольных выражения в исходном языке, θ — произвольная подстановка. Тогда

$$[\![f[x \leftarrow e]]\!]_\theta = [\![f]\!]_\theta[x \leftarrow [\![e]\!]_\theta]$$

В данном случае равенство необходимо трактовать как равенство двух множеств. Доказано структурной индукцией.

Определение 4. Для произвольной подстановки θ определим преобразование функционального контекста $[\![\bullet]\!]_\theta$ следующим образом:

$$
\begin{array}{rcl}
[\![\Box\, e]\!]_\theta & = & \Box\, [\![e]\!]_\theta \\
[\![v\, \Box]\!]_\theta & = & [\![v]\!]_\theta\, \Box \\
[\![\underline{\texttt{let}}\ x\ =\ \Box\ \underline{\texttt{in}}\ e]\!]_\theta & = & \underline{\texttt{let}}\ \ x\ =\ \Box\ \ \underline{\texttt{in}}\ \ [\![e]\!]_\theta
\end{array}
$$

Здесь e — произвольное функциональное выражение, v — лямбда-абстракция. Это преобразование является обобщением расширенного преобразования на случай функциональных контекстов, отсюда и схожее обозначение.

Определение 5. Для произвольной семантической переменной $\mathfrak{s}_1$, $\mathfrak{s}_2$ и произвольной подстановки θ определим преобразование атомарных контекстов $[\![\bullet]\!]_\theta^{\mathfrak{s}_1\mathfrak{s}_2}$ следующим образом:

$$
\begin{array}{l}
[\![C^k(v_1, \ldots, v_{i-1}, \Box, e_{i+1}, \ldots, e_k)]\!]_\theta^{\mathfrak{s}_1\mathfrak{s}_2} = \\
\quad \Box \wedge \\
\quad ([\![e_{i+1}]\!]_\theta\ \mathfrak{s}'_{i+1}) \wedge \\
\quad \ldots \\
\quad ([\![e_k]\!]_\theta\ \mathfrak{s}'_k) \wedge \\
\quad (\mathfrak{s}_2 \equiv\ C^k(\mathfrak{s}'_1, \ldots, \mathfrak{s}'_{i-1}, \mathfrak{s}_1, \mathfrak{s}'_{i+1}, \ldots, \mathfrak{s}_k)),\ \text{if}\ \theta(\mathfrak{s}'_j) = v_j,\ j < i
\end{array}
$$

$$
\begin{array}{rcl}
[\![\Box = e]\!]_\theta^{\mathfrak{s}_1\mathfrak{s}_2} & = & \Box \wedge \\
 & & ([\![e]\!]_\theta\ \mathfrak{s}') \wedge \\
 & & (((\mathfrak{s}_1 \equiv \mathfrak{s}') \wedge (\mathfrak{s}_2 \equiv \underline{\texttt{true}})) \vee \\
 & & ((\mathfrak{s}_1 \not\equiv \mathfrak{s}') \wedge (\mathfrak{s}_2 \equiv \underline{\texttt{false}})))
\end{array}
$$

$$
\begin{array}{rcl}
[\![v = \Box]\!]_\theta^{\mathfrak{s}_1\mathfrak{s}_2} & = & \Box \wedge \\
 & & (((\mathfrak{s}' \equiv \mathfrak{s}_1) \wedge (\mathfrak{s}_2 \equiv \underline{\texttt{true}})) \vee \\
 & & ((\mathfrak{s}' \not\equiv \mathfrak{s}_1) \wedge (\mathfrak{s}_2 \equiv \underline{\texttt{false}}))),\ \text{if}\ \theta(\mathfrak{s}) = v
\end{array}
$$

$$
\begin{array}{l}
[\![\underline{\texttt{match}}\ \Box\ \underline{\texttt{with}}\ \{C_i^{n_i}(y_1^i, \ldots, y_{n_i}^i) \to e_i\}]\!]_\theta^{\mathfrak{s}_1\mathfrak{s}_2} = \\
\quad \Box\ \wedge \bigvee_i \\
\quad (\underline{\texttt{fresh}}\ (s_1^i \ldots s_{n_i}^i) \\
\qquad (\mathfrak{s}_1 \equiv\ C_i^{n_i}(s_1^i, \ldots, s_{n_i}^i)) \\
\qquad (\lambda y_1^i. \ldots \lambda y_{n_i}^i. [\![e_i]\!]_\theta)\ (\equiv s_1^i) \ldots (\equiv s_{n_i}^i)\ \mathfrak{s}_2)
\end{array}
$$

Здесь $\mathfrak{s}'$ и $\mathfrak{s}'_i$ — произвольные семантические переменные, v_i — произвольные значения в исходном языке, e_i — произвольные выражения в исходном языке. Также потребуем, чтобы θ была неопределена для всех указанных семантических переменных, если явно не указано противоположное.

Определение 6. Для произвольной подстановки θ, произвольной семантической переменной $\mathfrak{s}_m$ и функционального выражения e опреде-

лим преобразования стека контекстов $[\![\bullet]\!]_\theta^{e,\mathfrak{s}_m}$ следующим образом:

$$[\![f_n \dots f_1 g_m \dots g_1]\!]_\theta^{e,\mathfrak{s}_m} = \begin{cases} [\![g_m]\!]_\theta^{\mathfrak{s}_m\mathfrak{s}_{m-1}} \dots [\![g_1]\!]_\theta^{\mathfrak{s}_1\mathfrak{s}_0}, \;\; n = 0 \;\text{ и } e \text{ — атомарное} \\ [\![f_n]\!]_\theta \dots [\![f_1]\!]_\theta(\square\, \mathfrak{s}_m)[\![g_m]\!]_\theta^{\mathfrak{s}_m\mathfrak{s}_{m-1}} \dots [\![g_1]\!]_\theta^{\mathfrak{s}_1\mathfrak{s}_0}, \text{ иначе} \end{cases}$$

Здесь $\mathfrak{s}_0 \dots \mathfrak{s}_{m-1}$ — произвольные уникальные семантические переменные.

Определение 7. Для произвольной подстановки θ и произвольной семантической переменной $\mathfrak{s}_m$ определим симуляционное преобразование $[\![\bullet]\!]_\theta^{\mathfrak{s}_m}$ для выражения исходного языка следующим образом:

$$\begin{aligned} [\![e_1 = e_2]\!]_\theta^{\mathfrak{s}_m} \;=\; & ([\![e_1]\!]_\theta\ \mathfrak{s}_1') \wedge \\ & ([\![e_2]\!]_\theta\ \mathfrak{s}_2') \wedge \\ & (((\mathfrak{s}_1' \equiv \mathfrak{s}_2') \wedge (\mathfrak{s}_m \equiv \underline{\texttt{true}})) \vee \\ & ((\mathfrak{s}_1' \not\equiv \mathfrak{s}_2') \wedge (\mathfrak{s}_m \equiv \underline{\texttt{false}}))) \\ [\![v = e]\!]_\theta^{\mathfrak{s}_m} \;=\; & ([\![e]\!]_\theta\ \mathfrak{s}_2') \wedge \\ & (((\mathfrak{s}_1' \equiv \mathfrak{s}_2') \wedge (\mathfrak{s}_m \equiv \underline{\texttt{true}})) \vee \\ & ((\mathfrak{s}_1' \not\equiv \mathfrak{s}_2') \wedge (\mathfrak{s}_m \equiv \underline{\texttt{false}}))),\ \text{if } \theta(\mathfrak{s}_1') = v \\ [\![v_1 = v_2]\!]_\theta^{\mathfrak{s}_m} \;=\; & (((\mathfrak{s}_1' \equiv \mathfrak{s}_2') \wedge (\mathfrak{s}_m \equiv \underline{\texttt{true}})) \vee \\ & ((\mathfrak{s}_1' \not\equiv \mathfrak{s}_2') \wedge (\mathfrak{s}_m \equiv \underline{\texttt{false}}))),\ \text{if } \theta(\mathfrak{s}_j') = v_j \end{aligned}$$

$$\begin{aligned} & [\![C^k(v_1, \dots, v_{i-1}, e_i, \dots, e_k)]\!]_\theta^{\mathfrak{s}_m} = \\ & \quad ([\![e_i]\!]_\theta\ \mathfrak{s}_i') \wedge \\ & \quad ([\![e_k]\!]_\theta\ \mathfrak{s}_k') \wedge \\ & \quad (\mathfrak{s}_m \equiv\ C^k(\mathfrak{s}_1', \dots, \mathfrak{s}_k')),\ \text{if } \theta(\mathfrak{s}_j') = v_j,\ j < i \end{aligned}$$

$$[\![C^k(v_1, \dots, v_k)]\!]_\theta^{\mathfrak{s}_m} = (\mathfrak{s}_m \equiv\ C^k(\mathfrak{s}_1', \dots, \mathfrak{s}_k')),\ \text{if } \theta(\mathfrak{s}_j') = v_j$$

$$[\![C^k(v_1, \dots, v_k)]\!]_\theta^{\mathfrak{s}_m} = (\mathfrak{s}_m \equiv\ \mathfrak{s}'),\ \text{if } \theta(\mathfrak{s}') = C^k(v_1, \dots, v_k)$$

$$\begin{aligned} & [\![\underline{\texttt{match}}\ e\ \underline{\texttt{with}}\ \{C_i^{n_i}(y_1^i, \dots, y_{n_i}^i) \to e_i\}]\!]_\theta^{\mathfrak{s}_m} = \\ & \quad [\![e]\!]_\theta\ \mathfrak{s}'\ \wedge\ \bigvee_i \\ & \quad (\underline{\texttt{fresh}}\ (s_1^i \dots s_{n_i}^i) \\ & \qquad (\mathfrak{s}' \equiv\ C_i^{n_i}(s_1^i, \dots, s_{n_i}^i)) \\ & \qquad (\lambda y_1^i. \dots \lambda y_{n_i}^i.[\![e_i]\!]_\theta)\ (\equiv s_1^i) \dots (\equiv s_{n_i}^i)\ \mathfrak{s}_m) \end{aligned}$$

$$[\![\underline{\texttt{match}}\ v\ \underline{\texttt{with}}\ \{C_i^{n_i}(y_1^i,\ ...,y_{n_i}^i) \to e_i\}]\!]_\theta^{\mathfrak{s}_m} =$$
$$\bigvee_i$$
$$(\underline{\texttt{fresh}}\ (s_1^i \ldots s_{n_i}^i)$$
$$(\mathfrak{s}' \equiv\ C_i^{n_i}(s_1^i, \ldots, s_{n_i}^i))$$
$$(\lambda y_1^i. \ldots \lambda y_{n_i}^i.[\![e_i]\!]_\theta)\ (\equiv s_1^i) \ldots (\equiv s_{n_i}^i)\ \mathfrak{s}_m),\ \text{if}\ \theta(\mathfrak{s}') = v$$

Здесь все $\mathfrak{s}'$ и $\mathfrak{s}'_i$ — произвольные семантические переменные, e — произвольное выражение, v — произвольное значение в оригинальной семантике. Также потребуем, чтобы θ была неопределена для всех указанных семантических переменных, если явно не указано противоположное.

Определение 8. Пусть

- $\langle \mathcal{S},\ e \rangle$ — состояние в откладвыющей семантике;
- $\left\langle \Sigma, \hat{\mathcal{S}}, \hat{e}, (\theta, \emptyset) \right\rangle$ —состояние ва реляционной семантике.

Назовём эти состояния **связанными**, если существует следующая семантическая переменная q_m.

- $\hat{\mathcal{S}} \in [\![\mathcal{S}]\!]_\theta^{e,\mathfrak{s}_m}$
- $\hat{e} \in \begin{cases} [\![e]\!]_\theta^{\mathfrak{s}_m} & , \begin{array}{l} e \text{ — атомарное,} \\ \mathcal{S} \text{ не содержит функциональных контекстов} \end{array} \\ [\![e]\!]_\theta & , \text{ иначе} \end{cases}$
- Σ содержит все семантические переменные из $\hat{e}$, $\hat{\mathcal{S}}$, и θ.

Лемма 4. Пусть $v = \mathtt{C}^k(v_1,...,v_k)$ – значение. Тогда для произвольных Σ, $\mathcal{S}$, θ, $\hat{v} \in [\![v]\!]_\theta$, и семантической переменной $\mathfrak{s}$ такой, что $\mathfrak{s} \notin dom(\theta)$ верно (5) или (6).

$$\langle \Sigma,\ \mathcal{S}, (\hat{v}\,\mathfrak{s}),\ (\theta,\ \emptyset) \rangle \rightsquigarrow^* \left\langle \Sigma',\ \mathcal{S},\ \mathfrak{s} \equiv \mathtt{C}^k(\mathfrak{s}'_1,...,\mathfrak{s}'_k),\ (\theta',\ \emptyset) \right\rangle \text{ и } \theta'(\mathfrak{s}'_i) = v_i \quad (5)$$

$$\langle \Sigma,\ \mathcal{S}, (\hat{v}\,\mathfrak{s}),\ (\theta,\ \emptyset) \rangle \rightsquigarrow \langle \Sigma,\ \mathcal{S},\ \mathfrak{s} \equiv \mathfrak{s}',\ (\theta,\ \emptyset) \rangle \text{ and } \theta(\mathfrak{s}') = v \quad (6)$$

Доказано индукцией по высоте v.

Лемма 5. Пусть $s = \langle \mathcal{S} = g_m \dots g_1,\, e\rangle$ — состояние в откладывающей семантике, g_i — атомарные контексты, e — выражение первого порядка, θ — некоторая подстановка, $\mathfrak{s}_m$ — некоторая семантическая переменная, $\hat{\mathcal{S}} \in [\![\mathcal{S}]\!]_\theta^{e,\mathfrak{s}_m}$, $\hat{e} \in [\![e]\!]_\theta$. Тогда существует последовательность шагов в реляционной семантике такая, что

$$\left\langle \Sigma, \hat{\mathcal{S}}, (\hat{e}\,\mathfrak{s}_m), (\theta,\, \emptyset)\right\rangle \rightsquigarrow^* \hat{s}$$

и s и $\hat{s}$ связаны. Это верно в предположении, что Σ содержит все семантические переменные из $\hat{\mathcal{S}}$ и θ. Доказательство разбором случаев для выражения e с использованием Леммы 4.

Лемма 6. Пусть $s_1 \to s_2$ — один шаг вычисления в откладывающией семантике, $\hat{s_1}$ — состояние в реляционной семантике такое, что s_1 и $\hat{s_1}$ связаны. Тогда существует последовательность шагов в реляционной семантике $\hat{s_1} \rightsquigarrow^* \hat{s_2}$ такая, что s_2 и $\hat{s_2}$ связаны.

Доказано методом разбором случаев для s_1 и конструктивным построением отношения симуляции [25; 26] с использованием Лемм 3, 4, 5.

Лемма 7. Пусть $s_0 = \langle \emptyset,\, \epsilon,\, \underline{\texttt{fresh}}\ (x)\ ([\![e]\!]^c\, x),\, \iota\rangle$ — начальное состояние в реляционной семантики. Тогда существует последовательность шагов $s_0 \rightsquigarrow^* \hat{s}$ такая, что $\langle \epsilon,\, e\rangle$ (начальное состояние в откладывающей семантике) и $\hat{s}$ связаны. Немедленно следует из Леммы 5.

Все необходимые определения и леммы сформулированы. Теперь докажем теорему о частичной динамической корректности. Пусть e – выражение первого порядка в исходном языке, которое вычисляется в значение $v = \mathtt{C}^k(v_1,...,v_k)$ в оригинальной семантике. Тогда выражение e вычисляется к этому же значинию в откладывающей семантике:

$$\langle \epsilon,\, e\rangle \to^* \langle \epsilon,\, v\rangle\,.$$

По Лемме 7 имеем

$$\langle \emptyset,\, \epsilon, \underline{\texttt{fresh}}\ (x)\ ([\![e]\!]^c\, x), \iota\rangle \rightsquigarrow^* \hat{s},$$

где $\langle \epsilon,\, e \rangle$ и $\hat{s}$ связаны. По Лемме 6 существует состояние $\hat{s'}$ в реляционной семантике такое, что

$$\hat{s} \rightsquigarrow^* \hat{s'},$$

где $\langle \epsilon,\, v \rangle$ и $\hat{s'}$ связаны. По определению отношения симуляции, $\hat{s'}$ представимо в форме (7) или (8).

$$\left\langle \Sigma,\, \epsilon,\, \mathfrak{s}_0 \equiv \mathtt{C}^k(\mathfrak{s}'_1,\ldots,\mathfrak{s}'_k),\, (\theta,\, \emptyset) \right\rangle,\, \theta(\mathfrak{s}'_i) = v_i \tag{7}$$

$$\left\langle \Sigma,\, \epsilon,\, \mathfrak{s}_0 \equiv \mathfrak{s}',\, (\theta,\, \emptyset) \right\rangle,\, \theta(\mathfrak{s}') = v \tag{8}$$

Здесь $\mathfrak{s}_0$ — первая семантическая переменная, введенная в Σ, и $\mathfrak{s}_0 \notin dom(\theta)$. В обоих случаях, остается сделать один последний шаг в реляционной семантике, который и завершает это доказательство.

7. Апробация

Описанный ранее метод преобразования функциональных программ в реляционные было реализовано на языке OCaml. В качестве входного языка используется подмножество OCaml, содержащее λ-исчисление, конструкторы, сопоставление с образцом, `let` (`rec`) связывания и полиморфное сравнение значений первого порядка.

При реализации были выявлены две проблемы. Во-первых, в результате преобразования присутствует множество абстракций, многие из которых могут быть применены немедленно. Для их устранения был добавлен дополнительный опциональный проход по абстрактному синтаксическому дереву, который выполняет β-редукции везде, где это возможно. Данная оптимизация значительно улучшает качество конвертируемых программ с точки зрения как удобочитаемости, так и производительности. Далее, в нашей первоначальной реализации слишком много значений преобразовывались в функции и, как результат, их тела вычисляются несколько раз с существенным ухудшением производительности. Нами была улучшена реализация путем определения важного конкретного случая и обработки его с небольшим преобразованием.

Отдельно стоит отметить, что в реализации метода преобразования используется встроенная в OCanren мемоизация отношений. Это позволяет значительно ускорить время работы рекурсивных функций.

В качестве первого примера преобразования рассмотрим реализацию функции конкатенации для списков (см. рисунок 10a). Результат преобразования (см. рисунок 10b) несколько отличается от классического реализации реляционного отношения конкатенации списков. Основное различие происходит от функционализации примитивных значений: в то время как обычные `append`^{o} работает со значениями-списками, преобразованный вариант использует функции, возвращающие цель. Таким образом, обычная `append`^{o} для аргументов `x`, `y` и `q` может быть выражено с помощью преобразованного в качестве `append`^{o} ($\equiv$ `x`) ($\equiv$ `y`) `q`.

Далее представлена демонстрация возможности выполнимости реляционных форм в различных направлениях на следующих примерах:

```
val append : α list →
             α list →
             α list
let rec append = λ a.λ b.
  match a with
  | Nil         → b
  | Cons (h, t) →
      Cons (h, append t b)
```

(a)

```
val append^o : ((α llist)^o → 𝔊) →
               ((α llist)^o → 𝔊) →
               (α llist)^o → 𝔊
let rec append^o a b q1 =
  fresh (q2)
    (a q2) ∧
    (((q2 ≡ Nil) ∧ (b q1)) |||
      (fresh (q3 q4)
        (q2 ≡ (Cons (q3, q4))) ∧
        (fresh (q6 q7)
          (q6 ≡ q3) ∧
          (q1 ≡ (Cons (q6, q7))) ∧
          (append^o (≡ q4) b q7))))
```

(b)

Рис. 10: Пример преобразование функции в отношение

- интерпретатор высшего порядка для лямбда-исчисления, принимающий в качестве аргумента функцию поиска подвыражения, к которому необходимо применить бета-редукцию;
- алгоритм Хиндли-Милнера [27] для вывода наиболее общего типа.

7.1. Интерпретатор высшего порядка для лямбда-исчисления

Как было сказано во введении, одним из применений языка miniKanren является разработка реляционных интерпретаторов [6; 9; 28]. Отличительной особенностью данного интерпретатора является его аргумент высшего порядка, который используется для поиска подвыражения, к которому применима бета-редукция. Таким образом, в зависимости от этого аргумента можно получить интерпретатор с различными стратегиями вычисления: *call-by-name*, *call-by-value*, нормальный порядок редукции и др. Реализованный функциональный интерпретатор и функции редукции имеют следующую сигнатуру:

```
val eval : (term → split) → term → term
val call_by_name  : term → split
val call_by_value : term → split
val normal_order  : term → split
```

где `term` – выражение лямбда-исчисления в нотации Де Брюэна; `split` – пара из выражения и контекста. Функция высшего порядка `eval` принимает в качестве первого аргумента функцию, определяющую порядок редукции, в качестве второго аргумента – выражение, которое необходимо редуцировать.

После преобразования будут получены отношения со следующими сигнатурами:

```
val eval^o : ((term^o → 𝔊) → split^o → 𝔊) → (term^o → 𝔊) →
             term^o → 𝔊
val call_by_name^o  : (term^o → 𝔊) → split^o → 𝔊
val call_by_value^o : (term^o → 𝔊) → split^o → 𝔊
val normal_order^o  : (term^o → 𝔊) → split^o → 𝔊
```

Полученное с помощью реляционного преобразования отношение позволяет непосредственно интерпретировать лямбда-выражения.

```
eval^o normal_order^o (≡ ‘(λ 0) 1‘) q ⇝ [q ↦ ‘1‘]
eval^o call_by_name^o (≡ ‘0 ((λ 0) 1)‘) q ⇝ [q ↦ ‘0 ((λ 0) 1)‘]
eval^o call_by_value^o (≡ ‘0 ((λ 0) 1)‘) q ⇝ [q ↦ ‘0 1‘]
```

В качестве примера рассмотрим три различных запроса с лямбда-выражениями и отношениями редукции. Во всех трех случаях запрос был успешно выполнен; для каждого лямбда-выражения была построена корректная нормальная форма, соответствующая выбранной стратегии редукции.

Также реляционная форма `eval`^{o} позволяет генерировать из нормальных форм (возможно, бесконечный) поток выражений, из которых эта нормальная форма была получена с помощью переданного отношения, определяющего стратегию вычисления.

```
eval^o normal_order^o (≡ q) (‘λ 0‘) ⇝ [
    q ↦ ‘λ 0‘;
    q ↦ ‘(λ 0) (λ 0)‘;
    q ↦ ‘λ ((λ 1) 0)‘;
    q ↦ ‘(λ 0) ((λ 0) (λ 0))‘; ...]
eval^o call_by_name^o (≡ q) (‘λ 0‘) ⇝ [
    q ↦ ‘λ 0‘;
    q ↦ ‘(λ 0) (λ 0)‘;
    q ↦ ‘(λ 0) ((λ 0) (λ 0))‘;
```

$\mathtt{q} \;\mapsto\; `(\lambda\ \lambda\, \mathbf{0})\ \ \mathbf{0}`;\ \ \ldots]$

В представленных выше запросах задана нормальная форма и стратегия редукции. Каждый из запросов порождает поток лямбда-выражений с заданной нормальной формой. В этом и последующих примерах результаты вычисления могут содержать свободные переменные, которые обозначаются числом в квадрате и интерпретируются как произвольное значение заданного типа.

Отметим, что аргумент, определяющий стратегию редукции, породить нельзя, так как он является функцией высшего порядка. Данное ограничение является следствием ограниченности синтаксической унификации.

7.2. Вывод типов Хиндли-Милнера

Данный алгоритм [27] по лямбда-выражению вычисляет наиболее общий тип этого выражения.

Функция `type_inference`, являющаяся реализацией алгоритма вывода типов и отношение $\mathtt{type\_inference}^o$, являющееся результатом преобразования `type_inference`, имеют следующие типы:

$\underline{\mathtt{val}}\ \mathtt{type\_inference}\ \ :\ \mathtt{term}\ \to\ \mathtt{typ}$
$\underline{\mathtt{val}}\ \mathtt{type\_inference}^o\ :\ (\mathtt{term}^o\ \to \mathfrak{G})\ \to \mathtt{typ}^o\ \to \mathfrak{G}$

Полученное с помощью реляционного преобразования отношение можно использовать непосредственно для вычисления типа выражения.

$\mathtt{type\_inference}^o\ (\equiv\ `\lambda x \to x`)\ \mathtt{q} \rightsquigarrow [\mathtt{q}\ \mapsto\ `a \to a`]$

Данный запрос для заданного выражения порождает корректный тип. Также реляционную форму $\mathtt{type\_inference}^o$ можно использовать для решения проблемы населенности типа.

$\mathtt{type\_inference}^o\ (\equiv \mathtt{q})\ \ 'a' \rightsquigarrow \bot$
$\mathtt{type\_inference}^o\ (\equiv \mathtt{q})\ \ 'a\ \to a' \rightsquigarrow [$
$\quad \mathtt{q}\ \mapsto\ '\lambda\ \boxed{0}\ \to \boxed{0}';$
$\quad \mathtt{q}\ \mapsto\ '\lambda\ \boxed{0}\ \to (\lambda\ \boxed{1}\ \to \boxed{1})\ \boxed{0}';$
$\quad \mathtt{q}\ \mapsto\ '\lambda\ \boxed{0}\ \to \underline{\mathtt{let}}\ \boxed{1} = \boxed{2}\ \underline{\mathtt{in}}\ \boxed{0}'\ (\boxed{0} \not\equiv \boxed{1});$
$\quad \mathtt{q}\ \mapsto\ '(\lambda\ \boxed{0}\ \to \boxed{0})\ \ (\lambda\ \boxed{1}\ \to \boxed{1})';\ \ \ldots]$

В первом запросе задан ненаселенный тип. Это подтверждает результат вычисления запроса отсутствием найденных выражений. Второй запрос породил бесконечный поток выражений, следовательно, заданный тип населен.

Наконец, данную реляционную форму можно использовать для достраивания выражения с дыркой таким образом, чтобы оно имело заданный тип.

$$\texttt{type\_inference}^o\ (\equiv\ `\underline{\texttt{let}}\ \texttt{f} = \square\ \underline{\texttt{in}}\ \texttt{f}\ (\lambda\ \texttt{x}\ \to \texttt{f}\ \texttt{x})`)\ \ `a \to a`\ \rightsquigarrow$$
$$[\square\ \mapsto\ `\lambda\ \boxed{0}\ \to \boxed{0}`;\ \ldots]$$

Данный запрос порождает выражения, которые можно подставить на место $\square$, после чего выражение будет корректно типизироваться.

7.3. Выводы по апробации

Прежде всего, апробация показала, что полученная с помощью описанного в данной работе метода преобразования реляционная форма функциональной программы может быть исполнена в прямом направлении. Другими словами, с помощью реляционной формы возможно вычислить результат исходной функции. Также реляционную программу можно использовать для вычисления произвольных аргументов первого порядка исходной функции. В случае неоднозначности этих аргументов будут вычислены все все подходящие значения этих аргументов, представленные в виде ленивого потока.

Однако с помощью реляционной формы нельзя вычислить аргумент высшего порядка вследствие ограничений синтаксической унификации, используемой в языке miniKanren.

8. Заключение

Подводя итог, были выделены следующие результаты:

1) описаны синтаксис, система вывода типов и операционная семантика для функционального и реляционного языков;

2) разработано преобразование типизированных функций в реляционную форму;

3) доказана статическая и динамическая корректность преобразования;

4) проведена апробация метода: реализован транслятор и исследована его применимость к нескольким классическим для реляционного программирования задачам;

5) данная работа была представлена на конференции Trends in Functional Programming 2017 и опубликована [29] в журнале Lecture Notes in Computer Science (Scopus).

Таким образом поставленная цель достигнута, а именно, разработан метод преобразования функциональных программ в реляционную форму.

Во многих случаях данный метод позволяет избежать трудоемкого “ручного” переписывания функциональных спецификаций в реляционную форму и сосредоточиться на разработке реляционных спецификациях только в том случае, когда их получение из функций невозможно или нежелательно.

Предложенный в данной работе метод преобразования применим к произвольным функциональным программам с ограниченным полиморфизмом. Это ограничение полиморфизма следует из ограничений языка miniKanren.

Апробация показала, что полученные с помощью метода преобразования реляционные формы можно исполнять для вычисления одного

или нескольких произвольных аргументов первого порядка. Более того, любой аргумент первого порядка можно определить частично, что позволяет гибко уточнять запрос к реляционной форме.

Список литературы

1. *Friedman D. P.*, *E.Byrd W.*, *Kiselyov O.* The Reasoned Schemer. — 2005.
2. Mercury language. — URL: `https://mercurylang.org` (дата обр. 10.11.2017).
3. Curry language. — URL: `http://www-ps.informatik.uni-kiel.de/currywiki` (дата обр. 10.11.2017).
4. miniKanren language. — URL: `http://minikanren.org` (дата обр. 10.11.2017).
5. *Hemann J.*, *Friedman D. P.* μKanren: A Minimal Functional Core for Relational Programming // Workshop on Scheme and Functional Programming. — 2013.
6. *Byrd W. E.* Relational Programming in miniKanren: Techniques, Applications, and Implementations. — Indiana University, Bloomington, 2009.
7. *Pierce B.* Types and Programming Languages. — MIT Press, 2002.
8. *Barendregt H. P.* Handbook of Logic in Computer Science (Vol. 2) // / под ред. S. Abramsky, D. M. Gabbay, S. E. Maibaum. — Oxford University Press, Inc., 1992. — Гл. Lambda Calculi with Types. С. 117—309.
9. *Byrd W. E.*, *Holk E.*, *Friedman D. P.* miniKanren, Live and Untagged: Quine Generation via Relational Interpreters (Programming Pearl) // Workshop on Scheme and Functional Programming. — 2012.
10. OCanren language. — URL: `http://github.com/dboulytchev/ocanren` (дата обр. 18.11.2017).
11. *Alvis C. E.*, *Willcock J. J.*, *Byrd W. E.* cKanren: miniKanren with Constraints // Workshop on Scheme and Functional Programming. — 2011.
12. *Baader F.*, *Snyder W.* Handbook of Automated Reasoning. — Elsevier, MIT Press, 2001.

13. *Кознов Д. В.* Методология и инструментарий предметно-ориентированного моделирования // Диссертация на соискание учёной степени доктора технических наук. — СПбГУ — 2016.

14. *Ольхович Л.*, *Кознов Д.* Метод автоматической валидации UML-спецификаций на основе OCL // Программирование. — 2003. — Т. 6. — С. 44—50.

15. RTST++: Methodology and a CASE Tool for the Development of Information Systems and Software For Real-Time Systems / A. N. Terekhov [и др.] // Programming and Computer Software. — 1999. — Т. 25. — С. 276—281.

16. Real: методология и CASE-средство для разработки систем реального времени и информационных систем / А. Н. Терехов [и др.] // Программирование. — 1999. — С. 44—51.

17. *Codognet P.*, *Diaz D.* WAMCC: Compiling Prolog to C // The MIT Press. — 1995. — С. 317—331.

18. *Henderson F.*, *Somogyi Z.* Compiling mercury to high-level C code // In Computational Complexity. — 2002. — С. 197—212.

19. *Banbara M.*, *Tamura N.*, *Inoue K.* Prolog Cafe: A prolog to Java translator system // Vol. 4369 of Lecture Notes in Computer Science. — 2006. — С. 1—11.

20. *Gómez-Zamalloa M.*, *Albert E.*, *Puebla G.* Decompilation of Java bytecode to Prolog by partial evaluation // Information and Software Technology. — 2009. — Т. 51, № 10. — С. 1409—1427. — Source Code Analysis and Manipulation, SCAM 2008.

21. *Calejo M.* InterProlog: Towards a Declarative Embedding of Logic Programming in Java.

22. *J. Cook J.* P#: A concurrent Prolog for the .NET framework.

23. *Wright A.*, *Felleisen M.* A Syntactic Approach to Type Soundness // Inf. Comput. — 1994. — Нояб. — Т. 115, № 1. — С. 38—94.

24. *Church A.*, *Rosser J. B.* Some Properties of Conversion // Transactions of the American Mathematical Society. — 1936. — Т. 39, № 3. — С. 472—482.

25. *N. A. Lynch F. W. V.* Forward and Backward Simulations: I. Untimed Systems // Inf. Comput. — 1995. — Т. 128, № 2. — С. 214—233.

26. *N. A. Lynch F. W. V.* Forward and Backward Simulations, II: Timing-Based Systems // Inf. Comput. — 1996. — Т. 128, № 1. — С. 1—25.

27. *Barendregt H.* Lambda Calculi with Types. — Handbook of Logic in Computer Science, Volume II, Oxford University Press, 1993.

28. A Unified Approach to Solving Seven Programming Problems (Functional Pearl) / W. E. Byrd [и др.] // Proc. ACM Program. Lang. — 2017.

29. *Lozov P.*, *Vyatkin A.*, *Boulytchev D.* Typed relational conversion // Lecture Notes in Computer Science. — 2018. — Т. 10788 LNCS. — С. 39—58.