Преобразование типизированных функций в реляционную форму

П.А. Лозов <lozov.peter@gmail.com>
Д.Ю. Булычев <dboulytchev@math.spbu.ru>
Санкт-Петербургский государственный университет,
198504, Россия, Санкт-Петербург, Университетский пр., д. 28

Аннотация. Реляционное программирование является подходом, позволяющим исполнять программы в различных "направлениях" для получения различных сценариев поведения по одной реляционной спецификации. В данной статье рассмотрена задача автоматического преобразования функциональных программ в реляционные. Представлен метод преобразования типизированных функций в реляционную форму, а также доказательство его статической и динамической корректности. Также в статье обсуждаются ограничения предложенного метода, представлена реализация метода для подмножества языка OCaml и проведена оценка эффективности метода на ряде реалистичных примеров.

Ключевые слова: функциональное программирование; реляционное программирование; генерация программ.

DOI: 10.15514/ISPRAS-2018-30(2)-3

Для цитирования: Лозов П.А., Булычев Д.Ю. Преобразование типизированных функций в реляционную форму. Труды ИСП РАН, том 30, вып. 2, 2018 г., стр. 45-64. DOI: 10.15514/ISPRAS-2018-30(2)-3

1. Введение

Реляционное программирование основано на построении программы в виде набора отношений [1]. 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, вычисление обратных функций. Хотя многие языки логического программирования, такие как Prolog, Mercury [2] или Curry [3], позволяют использовать некоторые реляционные эффекты, был создан miniKanren [4] – специальный язык для реляционного программирования. Изначально данный язык был небольшим предметно-ориентированным расширением языка Scheme/Racket, его минимальная реализация [5] содержала 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 определенных навыков. 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. Отметим, что единственный существующий подобный метод **Unnesting** [6] не полон и не был реализован.

Основным вкладом данной статьи 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].

Оставшаяся часть статьи организована следующим образом. В разд. 2 приведен краткий обзор схожих работ. Разд. 3 посвящена реляционному программированию и языку miniKanren. Разд. 4 содержит описание синтаксиса, правил вывода типов и операционной семантики как для ML-подобного языка, так и для его реляционного расширения. В разд. 5 представлены формальные правила для реляционного преобразования, а также доказательства сохранения типизации и семантики. В разд. 6 представлены реализация описанного метода и несколько примеров результата преобразования.

2. Схожие работы

Взаимодействие декларативных языков программирования (а реляционное программирование является декларативным) с языками более прагматичными (например, Java или OCaml) является естественной задачей. Успешное решение это задачи позволяет совместить выразительность декларативного языка с оптимизированностью и обширной функциональностью императивного или функционального языка. В качестве примера можно упомянуть работы [8; 9], где для описания ограничений на метамодель REAL [10] используется декларативный язык OCL (Object Constraint Language), который 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 является Prolog. Во-первых, между ним и языком miniKanren много общего; во-вторых, задача взаимодействия этого языка с другими хорошо исследована. Прежде всего, для языка Prolog существуют методы компиляции в более низкоуровневые языки, например в язык C [11; 12]. В [14] представлен метод преобразования из Prolog в Java, позволяющий увеличить эффективность исполнения декларативной программы, включающий метод декомпиляции Java Bitecode в Prolog [14]. В [15] представлен подход, встраивающий Prolog в Java, в [16] аналогично, Prolog встроен в C#. Тем не менее, не смотря на все эти

подходы, проблема взаимодействия декларативных и императивных языков программирования далека до полного разрешения.

В случае взаимодействия функционального языка с реляционным можно выделить обратную задачу: преобразование функциональных 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, исполнять функциональные программы реляционно (в частности, моделировать вычисление обратных функций). Для данной задачи было предложено решение Unnesting [6], которое, однако, рассматривает только случай нетипизированных программ и работает для специальных примеров. Более того, оно не был реализовано.

3. Реляционное программирование и язык 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 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” [17] –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 [18]. В контексте данной статьи будет использоваться конкретная реализация языка miniKanren – DSL на основе Objective Caml4, называемый OCanren [19]. Данный язык соответствует оригинальной реализации miniKanren [5], но OCanren дополнен конструкцией “Disequality constraint” [20].

Основной синтаксической единицей как реляционного языка miniKanren, так и языка OCanren является цель (*goal*). Для построения элементарных целей используется синтаксическая унификация (t1 ≡ t2) и обратная к ней операция disequality constraint ($t_1 \not\equiv t_2$). Для построения сложных целей используются дизъюнкция ($g_1 \vee g_2$), конъюнкция ($g_1 \wedge g_2$) и операция введения “свежей переменной” (**fresh** (x) g). Результат выполнения реляционной программы представляется в виде потока данных, из которого можно запрашивать решения. В качестве примера рассмотрим реляционную программу сложения чисел Пеано:


```
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'))
```

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). Также **a** можно представить 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 с должно быть на единицу больше, чем **c'** (строка 9).

```
fresh (x) add (S O) (S O) x ⇒ [x ≡ S (S O)]
(a)

fresh (x) add (S (S O)) x (S (S (S O))) ⇒ [x ≡ S O]
(b)

fresh (x y) add x y (S (S O)) ⇒ [ x ≡ O,     y ≡ S (S O);
                                  x ≡ S O,   y ≡ S O;
                                  x ≡ S (S O), y ≡ O ]
(c)

fresh (x) add (S(S (S O))) x (S (S O)) ⇒ []
(d)
```

Рис. 1. Примеры использования отношения **add**
Fig. 1. Examples of using relation **add**

Рассмотрим несколько различных целей, построенных с помощью отношения **add**, и изображенных на рис. 1. В каждом примере отношение принимает следующие возможные аргументы: выражения-константы, “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 рис. 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 рис. 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 рис. 1c приведён 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 рис. 1d приведен пример попытки прибавить к трем неотрицательное число и получить два.

4. Входной язык и его реляционное расширение

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. 2. Синтаксис входного языка
Fig. 2. Syntax of input language

Синтаксис исходного функционального языка показан на рис. 2. Данный язык является расширением лямбда-исчисления конструкторами с фиксированной размерностью C^n, двумя предопределенными конструкторами **true** и **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.

Данный язык оснащен системой вывода типов Хиндли-Милнера, правила которой описаны в прил. А. 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 **true** и **false** принадлежат выделенному алгебраическому типу **bool**.

Семантика данного языка представлена в прил. В и 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.

Наконец, выражение *e* вычисляется к результирующему значению *v* если

$$\langle \varepsilon, e\rangle \to^* \langle \varepsilon, v\rangle,$$

где ε – пустой стек, "$\to^*$" – рефлексивно-транзитивное замыкание отношения "$\to$".

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

Рис. 3. Синтаксис реляционного расширения
Fig. 3. Syntax of relational extension

Реляционное расширение добавляет пять стандартных конструкций языка miniKanren для построения целей, синтаксис которых отображен на рис. 3. Вследствие добавления конструкций miniKantren к конструкциям функционального языка, становится возможным построение всевозможных смешанных выражений, к примеру, конъюнкция (λx.x ∧ λy.y). Для устранения подобных некорректных выражений была расширена система типов для исходного языка, что описано в прил. С. 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 $\mathcal{J}$, олицетворяющим результат вычисления отношения.

Семантика расширенного языка представлена в прил. D. 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 **fresh**. Реляционное состояние используется при исполнении унификации и desequality constraint. Все существующие правила исходного языка дополняются множеством семантических переменных и реляционным состоянием, но не используют их.

5. Преобразование функциональных программ в реляционную форму

Прежде чем описать метод преобразования 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 представленного ниже преобразования в реляционную форму. Действительно, если **match**-выражение имеет тип высшего порядка, то его всегда можно преобразовать, используя η-расширение:

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

где $\bar{x}$ – это вектор новых переменных, отсутствующих в выражениях e, e_i и 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 $[\![\blacksquare]\!]^t$ определяется следующим образом:

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

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

Теперь рассмотрим преобразование выражений, обозначаемое $[\![\blacksquare]\!]^c$. Данное преобразование определеяется рекуррентно набором правил вида

$$[\![A]\!]^c = B,$$

где A – исходная функциональная программа, B – реляционная программа, являющаяся результатом преобразования. Ниже представлены правила для всех конструкций входного функционального языка.

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

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

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

В случае, если преобразовываемое выражение является конструктором, все его аргументы 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 с помощью унификации связываем переменную q с конструктором, примененным к созданным ранее переменным.

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

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_{ij} (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 окружим результат абстракциями, замыкающими все эти переменные и получим функцию. Теперь необходимо связать q_{ij} со свободными переменными из образца. Для этого применим описанную выше функцию к функциям, возвращающим цель ($\equiv q_{ij}$). В конечном итоге получим функцию, возвращающую цель, которую применим ко внешней переменной q, олицетворяющей результат исходного сопоставления с образцом.

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

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 без изменений.

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.

Для подтверждения корректности преобразования $\llbracket \blacksquare \rrbracket^c$ были сформулированы и доказаны следующие теоремы.

Теорема 1. (Статическая корректность). Если выражение e имеет тип *t* в исходном языке, тогда $\llbracket e \rrbracket^c$ с имеет тип $\llbracket t \rrbracket^t$ в реляционном расширении. Другими словами, преобразование в реляционную форму переводит правильно типизированные программы в правильно типизированные. Доказано с помощью структурной индукции.

Теорема 2. (Частичная семантическая корректность). Если выражение первого порядка *e* имеет тип *t* и *e* вычисляется до некоторого значения *v* $\left(e \xrightarrow{func} v\right)$, тогда $\underline{\texttt{fresh}}\ (x)\ (\llbracket e \rrbracket^c\ x)$ вычисляется до подстановки θ, и $\theta(\mathrm{v}) = v$, где v семантическая переменная, ассоциированная с *x* на первом шаге вычисления. Доказано с помощью симуляции.

6. Апробация

Описанное в предыдущем разделе метод преобразования функциональных программ в реляционные был реализован на языке OCaml. В качестве входного языка используется подмножество OCaml, описанное в разделе 4.

При реализации были выявлены две проблемы. Во-первых, результат преобразования содержит множество λ-абстракций, многие из которых могут быть применены немедленно. Для их устранения был добавлен дополнительный опциональный проход по абстрактному синтаксическому дереву преобразованной программы, который выполняет β-редукцию везде, где это возможно. Данная оптимизация значительно улучшает качество конвертируемых программ как с точки зрения читаемости, так и по производительности. Далее, в нашей первоначальной реализации слишком много значений преобразовывались в функции и, как результат, во время исполнения их тела вычислялись несколько раз с существенным ухудшением производительности. Нами была улучшена реализация путем внедрения в результирующую реляционную программу принудительного вычисления аргументов первого порядка для каждого содержащегося в программе вызова функции.

В качестве первого примера преобразования рассмотрим реализацию функции конкатенации для списков (см. рис. 4a). Результат преобразования (рис. 4b) несколько отличается от классического реализации реляционного отношения

конкатенации списков. Основное различие происходит от функционализации примитивных значений: в то время как обычные **append°** работает со значениями-списками, преобразованный вариант использует функции, возвращающие цель. Таким образом, классическая **append°** для аргументов **x**, **y** и **q** может быть выражено с помощью преобразованного в качестве **append° (≡ x) (≡ y) q**.

Далее мы продемонстрируем возможность выполнимости реляционных форм в различных направлениях на следующих примерах:

- интерпретатор высшего порядка для лямбда-исчисления, принимающий в качестве аргумента функцию поиска подвыражения, к которому необходимо применить бета-редукцию;
- алгоритм Хиндли-Милнера [7] для вывода наиболее общего типа.

```
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° : (α llist → G) →
              (α llist → G) →
              α llist → G
let rec append° 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° (≡ q4) b q7))))
```

(b)

Рис. 4. Пример преобразования функции в отношение
Pic. 4. Example of relational conversion

6.1 Интерпретатор высшего порядка для лямбда-исчисления

Как было сказано во введении, одним из применений языка miniKanren является разработка реляционных интерпретаторов [6; 18; 21]. 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 редуциии 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° : ((term → G) → split → G) → (term → G) →
            term → G
val call_by_name°  : (term → G) → split → G
val call_by_value° : (term → G) → split → G
val normal_order°  : (term → G) → split → G
```

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

```
eval° normal_order° (≡ ‘(λ 0) 1‘) q ⇝ [q ↦ ‘1‘]
eval° call_by_name° (≡ ‘0 ((λ 0) 1)‘) q ⇝ [q ↦ ‘0 ((λ 0) 1)‘]
eval° call_by_value° (≡ ‘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°** позволяет генерировать из нормальных форм (возможно, бесконечный) поток выражений, из которых эта нормальная форма была получена с помощью переданного отношения, определяющего стратегию вычисления.

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

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.

6.2 Вывод типов Хиндли-Милнера

Данный алгоритм [22] по лямбда-выражению вычисляет наиболее общий тип этого выражения.

Функция **type_inference**, являющаяся реализацией алгоритма вывода типов и отношение **type_inference°**, являющееся результатом преобразования **type_inference**, имеют следующие типы:

```
val type_inference  : term → typ
val type_inference° : (term → 𝔅) → typ → 𝔅
```

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

```
type_inference° (≡ ‘λx → x‘) q ⇝ [q ↦ ‘a → a‘]
```

Данный запрос для заданного выражения порождает корректный тип. Также реляционную форму **type_inference°** можно использовать для решения проблемы населенности типа.

```
type_inference° (≡ q) 'a' ⇝ ⊥
type_inference° (≡ q) 'a → a' ⇝ [
    q ↦ 'λ 0 → 0';
    q ↦ 'λ 0 → (λ 1 → 1) 0';
    q ↦ 'λ 0 → let 1 = 2 in 0' (0 ≠ 1);
    q ↦ '(λ 0 → 0) (λ 1 → 1)'; ...]
```

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.

```
type_inference° (≡ ‘let f = □ in f (λ x → f x)‘) ‘a → a‘ ⇝
  [□ ↦ ‘λ 0 → 0‘; ...]
```

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

7. Выводы

В данной работе представлен метод для преобразования типизированных функциональных программ в реляционную форму. Во многих случаях данное преобразование позволяет избежать утомительного переписывания функциональных спецификаций в реляционную форму и сосредоточиться на

реляционных спецификациях только тогда, когда их получение из функций невозможно или нежелательно.

Наш метод преобразования применим только к функциональным программам с ограниченным полиморфизмом, вследствие ограничений синтаксической унификации, используемой в языке miniKanren.

Апробация показала, что полученные с помощью метода преобразования реляционные формы можно исполнять для вычисления произвольного аргумента первого порядка. Более того, любой аргумент первого прядка 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. MIT Press, 2005.

[2]. Язык Mercury. URL: https://mercurylang.org (дата обращения 09.04.2018).

[3]. Язык Curry. URL: http://www-ps.informatik.uni-kiel.de/currywiki (дата обращения 09.04.2018).

[4]. Язык miniKanren. URL: http://minikanren.org (дата обращения 09.04.2018).

[5]. Hemann J., Friedman D. P. µKanren: A Minimal Core for Relational Programming. Workshop on Scheme and Functional Programming, 2013.

[6]. Byrd W. E. Relational Programming in miniKanren: Techniques, Applications, and Implementations. Ph.D. thesis, Indiana University, Bloomington, 2009.

[7]. Pierce B. Types and Programming Languages. MIT Press, 2002.

[8]. Кознов Д. В. Методология и инструментарий предметно-ориентированного моделирования. Диссертация на соискание учёной степени доктора технических наук, СПбГУ, 2016.

[9]. Ольхович Л., Кознов Д. Метод автоматической валидации UML-спецификаций на основе OCL. Программирование, 2003, том 29, № 6, стр. 44–50.

[10]. Терехов А.Н., Романовский К.Ю., Кознов Д.В., Долгов П.С., Иванов А.Н.. RTST++: методология и CASE-средство для разработки информационных систем и программного обеспечения для систем реального времени. Программирование, 1999, том 25, № 5.

[11]. Codognet P., Diaz D. WAMCC: Compiling Prolog to C. The MIT Press, 1995, pp. 317–331.

[12]. Henderson F., Somogyi Z. Compiling mercury to high-level C code. In Computational Complexity, 2002, pp. 197–212.

[13]. Banbara M., Tamura N., Inoue K. Prolog Cafe: A prolog to Java translator system. Lecture Notes in Computer Science, vol. 4369, 2006, pp. 1–11.

[14]. G´omez-Zamalloa M., Albert E., Puebla G. Decompilation of Java bytecode to Prolog by partial evaluation. Information and Software Technology, 2009, vol. 51, № 10, pp. 1409–1427.

[15]. Calejo M. InterProlog: Towards a Declarative Embedding of Logic Programming in Java. JELIA 2004: Logics in Artificial Intelligence, pp. 714-717.

[16]. J. Cook J. P#: A concurrent Prolog for the .NET framework. Software Practice and Experience, vol. 34. № 9, 2004, pp. 815-845.

[17]. 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.
[18]. Kosarev D., Boulytchev D. Typed Embedding of a Relational Language in OCaml. ACM SIGPLAN Workshop on ML, 2016.
[19]. Язык OCanren. URL: http://github.com/dboulytchev/ocanren (дата обращения 09.04.2018).
[20]. Alvis C. E., Willcock J. J., Byrd W. E. cKanren: miniKanren with Constraints, Workshop on Scheme and Functional Programming, 2011.
[21]. Byrd W. E., Ballantyne M., Rosenblatt G., Might M. A Unified Approach to Solving Seven Programming Problems (Functional Pearl). Proc. ACM Program. Lang, 2017, vol. 1, ICFP, pp. 8:1–8:26.
[22]. Barendregt H. Lambda Calculi with Types. Handbook of Logic in Computer Science, Volume II, Oxford University Press, 1993.

Приложения

A. Правила типизации для входного языка

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:

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

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

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

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

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

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

B. Семантика входного языка

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 сотояние); $\langle \epsilon, v \rangle$ (финальное состояние)

Переходы:

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

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

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

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

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

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

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

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

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

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

C. Правила типизации для реляционного расширения

Типы:

$$\begin{array}{lcll} \mathcal{L} & = & \alpha^o \mid (T^n(l_1, \ldots, l_n))^o & \text{(логические переменные)} \\ \mathcal{T} & += & \mathfrak{G} & \end{array}$$

Правила типизации:

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

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

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

D. Семантика для реляционного расширения

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 семмантических переменных, стек контекстов, выражение, логическое состояние)

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

Переходы:

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

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

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

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

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

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

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

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

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

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

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

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

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

Conversion Typed Functions into Relational Form

P. Lozov <lozov.peter@gmail.com>
D. Boulytchev <dboulytchev@math.spbu.ru>
St. Petersburg State University,
Universitetski pr., 28, 198504 St. Petersburg, Russia

Abstract. Relational programming is an approach that allows you to execute programs in different "directions" to get different behaviors from one relational specification. The direct development of relational programs is a complex task, requiring the developer to have certain skills. However, in many cases, the required relational program can be obtained from a certain functional program automatically. In this paper, the problem of automatic conversion of

functional programs into relational ones is considered. The main contribution of the paper is the method of converting typed functions into a relational form, as well as proving its static and dynamic correctness. This method can be applied to typed programs of a general kind. To describe these programs, a compact ML-like language (a subset of OCaml) is used, equipped with a Hindley-Milner type system with let-polymorphism Also, the paper discusses the limitations of the proposed method, presents an implementation for a subset of the OCaml language, and evaluates the method on a number of realistic examples.

Keywords: functional programming; relational programming; conversion of programs.

DOI: 10.15514/ISPRAS-2018-30(2)-3

For citation: Lozov P., Boulytchev D. Conversion Typed Functions into Relational Form. Trudy ISP RAN/Proc. ISP RAS, vol. 30, issue 2, 2018, pp. 45-64 (in Russian). DOI: 10.15514/ISPRAS-2018-30(2)-3

References

[1]. Friedman D. P., E.Byrd W., Kiselyov O. The Reasoned Schemer. MIT Press, 2005.
[2]. Mercury Language. URL: https://mercurylang.org (accessed 09.04.2018).
[3]. Curry Language. URL: http://www-ps.informatik.uni-kiel.de/currywiki (дата обращения 09.04.2018).
[4]. miniKanren Language. URL: http://minikanren.org (accessed 09.04.2018).
[5]. Hemann J., Friedman D. P. μKanren: A Minimal Core for Relational Programming. Workshop on Scheme and Functional Programming, 2013.
[6]. Byrd W. E. Relational Programming in miniKanren: Techniques, Applications, and Implementations. Ph.D. thesis, Indiana University, Bloomington, 2009.
[7]. Pierce B. Types and Programming Languages. MIT Press, 2002.
[8]. Koznov D. V. Methodology and tools for object-oriented modeling. PhD Thesis, SPBU, 2016 (in Russian).
[9]. Ol'khovich L., Koznov D.V. Ocl-based Automated Validation Method For Uml Specifications. Programming and Computer Software, 2003, vol. 29, № 6, pp. 323–327. DOI: 10.1023/B:PACS.0000004132.42846.11.
[10]. Terekhov A. N., Romanovskii K. Yu., Koznov D. V., Dolgov P. S., Ivanov A. N. RTST++: Methodology and a CASE Tool for the Development of Information Systems and Software For Real-Time Systems. Programming and Computer Software, 1999, vol. 25, № 5, pp. 276–281.
[11]. Codognet P., Diaz D. WAMCC: Compiling Prolog to C. The MIT Press, 1995, pp. 317–331.
[12]. Henderson F., Somogyi Z. Compiling mercury to high-level C code. In Computational Complexity, 2002, pp. 197–212.
[13]. Banbara M., Tamura N., Inoue K. Prolog Cafe: A prolog to Java translator system. Lecture Notes in Computer Science, vol. 4369, 2006, pp. 1–11.
[14]. G´omez-Zamalloa M., Albert E., Puebla G. Decompilation of Java bytecode to Prolog by partial evaluation. Information and Software Technology, 2009, vol. 51, № 10, pp. 1409–1427.
[15]. Calejo M. InterProlog: Towards a Declarative Embedding of Logic Programming in Java. JELIA 2004: Logics in Artificial Intelligence, pp. 714-717.

[16]. J. Cook J. P#: A concurrent Prolog for the .NET framework. Software Practice and Experience, vol. 34. № 9, 2004, pp. 815-845.
[17]. 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.
[18]. Kosarev D., Boulytchev D. Typed Embedding of a Relational Language in OCaml. ACM SIGPLAN Workshop on ML, 2016.
[19]. Язык OCanren. URL: http://github.com/dboulytchev/ocanren (accessed 09.04.2018).
[20]. Alvis C. E., Willcock J. J., Byrd W. E. cKanren: miniKanren with Constraints, Workshop on Scheme and Functional Programming, 2011.
[21]. Byrd W. E., Ballantyne M., Rosenblatt G., Might M. A Unified Approach to Solving Seven Programming Problems (Functional Pearl). Proc. ACM Program. Lang, 2017, vol. 1, ICFP, pp. 8:1–8:26.
[22]. Barendregt H. Lambda Calculi with Types. Handbook of Logic in Computer Science, Volume II, Oxford University Press, 1993.