DOI: 10.15514/ISPRAS-2019-1(2)-1

Строго типизированное встраивание реляционного языка программирования в OCaml

[1] *Д.С. Косарев,* ORCID: 0000-0002-6773-5322 <*d.kosarev@spbu.ru*>
[1] *Д.Ю. Булычев,* ORCID: 0000-0001-8363-7143 <*dboulytchev@math.spbu.ru*>
[1] *Санкт-Петербургский государственный университет,*
Россия, 198504, Санкт-Петербург, Университетский пр., д. 28.

Аннотация. В работе представлена реализация реляционного языка программирования miniKanren, встроенного в функциональный язык OCaml. Отличительная особенность описываемой реализации — использование *полиморфной унификации*, которая может использоваться для унификации произвольных типов данных. Также представлен систематический подход к разработке реляционных программ и интеграции их с функциональным кодом. Реализация включает в себя стандартное расширение реляционного программирования — ограничения-неравенства, также мы предлагаем использовать шаблонные логические переменные для увеличения выразительной силы ограничений-неравенств.

Ключевые слова: синтез программ, программирование в ограничениях, реляционное программирование, miniKanren.

Для цитирования: Косарев Д.С., Булычев Д.Ю. Строго типизированное встраивание реляционного языка программирования в OCaml. Труды ИСП РАН, том 1, вып. 2, 2019 г., стр. 15–19. DOI: 10.15514/ISPRAS–2019–1(2)–1.

Strongly Typed Emebedding of Relational Language in OCaml

[1] *D.S. Kosarev* ORCID: 0000-0002-6773-5322 <*d.kosarev@spbu.ru*>
[1] *D.Yu. Boulytchev* ORCID: 0000-0001-8363-7143 <*dboulytchev@math.spbu.ru*>
[1] *St. Petersburg State University,*
Universitetski pr., 28, St. Petersburg, 198504, Russia.

Abstract. We propose an implementation of relational language miniKanren embedded into strongly typed functional language OCaml. The peculiarity of our implementation — *polymorphic unification* — allows us to construct unification algorithm for arbitrary data types. Also, we provide a systematic approach for relational code construction and its integration with general purpose functional code. Our implementation incorporates standard extension of relational programming — disequality constraint, also we propose *wildcard logic variables* which add more expressivity to disequality constraints.

Keywords: program synthesis, constraint programming, relational programming, miniKanren.

For citation: Kosarev D.S., Boulytchev D.Yu. Strongly Typed Emebedding of Relational Language in OCaml . *Trudy ISP RAN/Proc. ISP RAS*, vol. 1, issue 2, 2019. pp. 15-19 (in Russian). DOI: 10.15514/ISPRAS-2019-1(2)-1.

1. Введение

Реляционное программирование [1] — это привлекательный подход, основанный на представлении программ как отношений. В результате реляционные программы можно “запускать” в различных “направлениях”, что позволяет, например, симулировать обратимые вычисления. Это интересно не только с теоретической точки зрения, но обладает и практической ценностью: некоторые задачи выглядят проще, если их рассматривать как запросы к некоторой реляционной спецификации [2]. Можно назвать некоторое количество примеров, которые подтверждают это наблюдение: алгоритм проверки типов для просто типизированного λ-исчисления (и в то же время алгоритм вывода типов и проверки населенности типов); интерпретатор, способный синтезировать “квайны” — программы, вычисляющиеся в самих себя; сортировка списков, способная породить все перестановки, т.п.

Многие языки логического программирования (например, Prolog, Mercury [3] или Curry [4]) в некоторой степени могут считаться реляционными. Нами был выбран miniKanren[1] как модельный язык, потому что он был специально спроектирован как предметно-ориентированный язык, встраиваемый в Scheme/Racket. Он довольно минималистичен — может быть реализован с помощью малого количества структур данных и комбинаторов [5, 6] — и поэтому можно найти реализации, где его уже реализовали для других языков программирования общего назначения, включая Scala, Haskell и Standard ML. Парадигму в основе miniKanren можно описать как “легковесное логическое программирование”[2].

В данной работе исследуется задача встраивания miniKanren в OCaml[3] — статически типизированный функциональный язык с богатой системой типов. Статическая типизация приносит ряд улучшений. Во-первых, типизация даёт некоторый уровень корректности, отсеивая патологические программы, выдающие патологические результаты. В контексте реляционного программирования типизация также помогает интерпретировать результаты запросов. Часто ответы на реляционные запросы содержат свободные переменные, которые можно заменять на произвольные значения. В типизированном случае эти переменные становятся типизированными, что упрощает понимание ответов, особенно с большим количеством свободных переменных. Во-вторых, некоторые программы на miniKanren требуют реализации и использования специальных ограничений. В слабо типизированном случае, где всё может быть чем угодно, некоторые символы и структуры данных могут “просачиваться” в нежелательные места [7]. Чтобы это предотвращать в miniKanren для Scheme были добавлены дополнительные ограничения (“absento”, “symbolo”). Эти ограничения важны для слабых динамических систем типов, так как они отсекают ответы во время исполнения. В статически типизированных языках эта работа может быть переложена на систему типов (при условии, что программист реализует их правильно), что не только улучшит производительность, но и уменьшит количество примитивов miniKanren, которые требуется реализовать.

В работе представлена реализация[4] реляционных комбинаторов и синтаксических расширений для языка OCaml под названием OCanren, которая в том числе поддерживает ограничения-неравенства [8] и оптимизации faster-miniKanren[5], которые применяются в наиболее полных реализациях. Новизна заключается в следующем:

[1] http://minikanren.org
[2] Детальное сравнение miniKanren и Prolog можно найти здесь: http://minikanren.org/minikanren-and-prolog.html (проверено: 10 января 2025).
[3] http://ocaml.org
[4] https://github.com/PLTools/OCanren
[5] https://github.com/michaelballantyne/faster-minikanren

- Реализованное встраивание позволяет программисту использовать статическую типизацию и вывод типов в реляционных спецификациях. В частности, ошибки типизации находятся во время компиляции и типы логических переменных выводятся из контекста.
- Реализация основана на полиморфной унификации, которая также как и полиморфное сравнение языка OCaml, может использоваться для произвольных типов. Реализация полиморфной унификации использует небезопасные возможности компилятора OCaml и полагается на специфичные особенности представления значений в памяти. При этом доказано, что корректность типов не нарушается.
- Описан масштабируемый подход (глава 9) к использованию типов при написании реляционных спецификаций, которые позволяет конвертировать данные из реляционного домена и в него. Подход основан на обобщённом программировании и позволяет более кратко и единообразно писать реляционные спецификации.
- Представлен упрощенный способ интеграции реляционного и функционального кода. Он основан на известном методе [9, 10] реализации поливариадических функций, и позволяет скрыть процесс реификации (глава 7) от программиста-пользователя.

В разделе 2 представлен обзор. Затем мы кратко описываем основные конструкции miniKanren в его оригинальном виде и вводим некоторые понятия реляционного программирования. Изложения языка со всеми деталями не предполагается, интересующиеся читатели могут обратиться к книге [1] по реляционному программированию. В разделе 4 описываются необходимые конструкции для реализации реляционного языка, в это раз в контексте OCaml. В разделе 5 вводится полиморфная унификация, а также показано, что этот алгоритм унификации с использованием треугольной подстановки не нарушает корректности типов. Затем представлены возможности по поддержке пользовательских типов данных путём их инъекции в логический домен, а также шаблон обобщённого программирования, который позволяет реализовывать преобразования в логический домен и обратно. Также представлен (раздел 6) более сложный подход, где данные представляются без тегирования ради эффективности. В разделе 7 описаны конструкции верхнего уровня и обсуждается задача интеграции функционального и реляционного кода. Затем представлены несколько примеров (раздел 9) написания реляционных программ, с помощью нашей библиотеки. В разделе 10 приведены результаты оценки производительности нашего подхода и реализаций miniKanren на языке Scheme.

2. Обзор

При реализации miniKanren в строго статически типизированном языке естественным образом возникают сложности, так как реляционное программирование родилось в среде Scheme/Racket — динамически типизированной и удобной для метапрограммирования среде. Оригинальная реализация miniKanren не обращает внимания на особенности, важные для таких языков как ML и Haskell. Например, унификация, одна из основных конструкций miniKanren, обязана исполняться для разных данных, и одного только параметрического полиморфизма может быть недостаточно.

Существуют несколько способов преодолеть эту проблему. Во-первых, можно смоделировать нетипизированный подход и предоставить реализацию унификации для некоторого конкретного типа, достаточно богатого, чтобы представить все другие типы

данных. Некоторые библиотеки[6] реляционного программирования для Haskell и предыдущая[7] реализация для OCaml пошли по этому пути. В результате оригинальная реализация на Scheme может быть повторена со всей её краткостью, но реляционные спецификации станут слабо типизированы. Сходный подход встречался ранее при встраивании Prolog в Haskell [11].

Другим подходом является использование *ad hoc* полиморфизма и написание специфичной унификации для каждого из интересующих типов. Некоторые реализации miniKanren, например Molog[8] and MiniKanrenT[9] для Haskell, выбрали данный подход. Строгая типизация сохраняется, но требуется написание большого количества стереотипного (англ. boilerplate) кода; часто применяется некоторая оптимизация этого, например с помощью Template Haskell [12]. Можно определять [13] отдельные классы типов для выполнения и унификации, и выделения свободных логических переменных. Для нас выглядит несколько искусственно требование выделять логические переменные в пользовательских типах данных особым способом, так как эти логические переменные придется обрабатывать и в модулях программы, не имеющих отношения к логическому программированию.

Существует ещё один подход, но нам неизвестны реализации miniKanren на его основе. Можно реализовать унификацию для обобщённого представления типов данных в виде сумм произведений и неподвижных точек функторов для них [14, 15]. Унификация сможет работать для любого типа данных, для которого сделано представление. Мы полагаем, что с таким подходом будет меньше стереотипного кода.

Из выше сказанного можно заключить, что типизированное встраивание miniKanren в OCaml состоит из сочетания обобщённого программирования [16] и *ad hoc* полиморфизма.

Использования обобщённого программирования в OCaml уже довольно хорошо исследовано [17]. Возможности для *ad hoc* полиморфизма в OCaml развиты слабо: нет ничего сравнимого с классами типов языка Haskell, использование объектно-ориентированного программирования позволяет эмулировать желаемое поведение, но не без недостатков. Существующие предложения по развитию *ad hoc* полиморфизма (например, неявные модули [18]) требуют модификации компилятора, чего мы хотели бы избежать. Поэтому мы пошли другим путём и реализовали полиморфную унификацию один раз и для всех логических типов, и эта реализация существенно использует особенности компилятора OCaml, так как возможности для менее *ad hoc* подхода пока не интегрированы в язык. Для работы с пользовательскими типами данных в реляционных частях программ предлагается использовать специальные логические представления (глава 6), которые освобождают пользователя от задачи поддержки переменных в его типах данных. Использование же обобщённого программирования позволяет систематическим образом получать преобразования в логические представления и обратно.

3. miniKanren. Краткое изложение

В этом разделе miniKanren будет кратко описан в его оригинальном виде на основе канонических примеров. Предметно-ориентированный язык организован как набор комбинаторов и макросов для Scheme/Racket, которые должны описывать поиск решения для некоторой “цели” (англ. goal). Ядро языка состоит из четырех реляционных конструкции для создания целей:

[6] https://github.com/JaimieMurdock/HK, https://github.com/rntz/ukanren (проверено 19 июня 2025)
[7] https://github.com/lightyang/minikanren-ocaml
[8] https://github.com/acfoltzer/Molog
[9] https://github.com/jvranish/MiniKanrenT

- Ограничение синтаксической унификации [19] в форме (== t_1 t_2), где t_1, t_2 — это некоторые “термы”. Унификация является основой для проведения поиска: если существует унификатор двух термов, то цель считается выполненной, наиболее общий унификатор сохраняется как частичное решение, и поиск в данной ветви дерева поиска.
- Ограничения-неравенства [8] в форме (≠ t_1 t_2), где t_1, t_2 — это некоторые “термы”. Данное ограничение запрещает поиск в поддеревьях, где данные темы равны с учетом соответствующей подстановки.
- Условная конструкция в форме conde, где каждый g_{ij} является некоторой целью. Построенная с использованием conde цель, рассматривает коллекцию подцелей в квадратных скобках как неявную конъюнкцию (т.е. [g_{i1} g_{i2} … g_{ik}] считается конъюнкцией всех g_{ij}) и пытается решать их независимо, т.е. conde работает как дизъюнкция.

```
(conde
  [g11 g12 … g1k1 ]
  [g21 g22 … g2k2 ]
  …
  [gn1 gn2 … gnkn ])
```

- Создание свежих (англ. fresh) логических переменных осуществляется с помощью Конструкции fresh, где каждый g_i является некоторой целью. Создаются новые свежие логические переменные x_1 x_2 … x_k и осуществляется поиск цели, являющейся конъюнкцией целей g_1 … g_n, внутри которых могут использоваться только что введенные логические переменные.

```
(fresh (x1 x2 … xk )
  g1
  …
  gn)
```

В качестве примера рассмотрим отношение append^o для конкатенации списков (по традиции реляционные объекты именуются с суффиксом^o).

```
(define (appendº x y xy)
  (conde
    [(== '() x) (== y xy)]
    [(fresh (h t ty)
       (== `(,h . ,t) x)
       (== `(,h . ,ty) xy)
       (appendº t y ty))]))
```

Рис. 6. Реляционная реализация сложения чисел Пеано в оригинальном miniKanren
Pic. 6. Relational implementation of Peano numbers addition using original miniKanren

Отношение “append^o x y xy” интерпретируется как “конкатенация x и y даёт xy”. Если список x пуст (строка 3), то, вне зависимости от содержимого y, чтобы отношение выполнялось значение xy должно быть таким же как y. Иначе, x нужно разделить на голову h и хвост t, для этого понадобятся введённые свежие переменные. Также пригодится

дополнительная переменная ty, чтобы хранить список, находящий в отношении appendº с t и y. Дополнив тривиальными соображениями (строки 5–7), получим полную реализацию отношения.

Цели, полученные с помощью упомянутых выше конструкций, можно запускать с помощью примитива run:

$$\texttt{run}\ n\ (q_1\ \ldots\ q_k)\ G.$$

Здесь n — это число желаемых ответов (для получения всех ответов используется “*”), q_i — свежие искомые переменные, а G — цель, использующая эти переменные.

Конструкция run осуществляет поиск ответов для данной цели, и возвращает потенциально бесконечный список ответов, состоящий из значений искомых переменных, которые хранят ответы на данный запрос. Например

```
run 1 (q) (appendº q '(3 4) '(1 2 3 4))
```

возвращает список `((1 2))`, который хранит одно значение искомой переменной q. Процесс реконструирования ответов из внутреннего представления называет *реификацией*[10][2].

4. Потоки, состояния и цели

Этот раздел содержит некоторые детали реализации. Несмотря на то, что это почти повторение оригинальной реализации [5, 8] для OCaml, он нужен, чтобы аккуратно ввести некоторые понятия.

Процедура поиска реализована с использованием монады для ленивых потоков с возвратами [20]:

```
type α stream
val mplus: α stream → α stream → α stream
val bind: α stream → (α → β stream) → β stream
```

Важной компонентой является реализация следующих типов данных.

```
type env = ...
type subst = ...
type constraints = ...
type state = env * subst * constraint
```

Тип состояния state описывает позицию в лениво строящемся дереве поиска: тип env соответствует *окружению*, содержащему дополнительную информацию (в частности, необходимую для создания свежих переменных); subst хранит подстановку, которая содержит отображения некоторых логических переменных; тип constraints представляет ограничения-неравенства, которые надо соблюдать (раздел 8). В самом простом случае env содержит только номер последней введённой свежей переменной, subst — это конечное отображение, а constraints — просто список подстановок.

Тип целей goal является преобразованием одного состояния в ленивый поток состояний:

[10] Реификация (англ. reification) — превращение чего-то абстрактного (т.е. существующего в виде идеи) во что-то конкретное. Кембриджский словарь. https://dictionary.cambridge.org/dictionary/english/reification (проверено 28 января 2025 г.)

```
type goal = state → state stream
```

В терминах поиска выполнение цели соответствует одному шагу поиска: для конкретного узла дерева поиска вычисляются его непосредственные потомки. Со стороны пользователя тип goal является абстрактным и все состояния полностью скрыты.

Также присутствует набор заранее реализованных комбинаторов:

```
val (&&&): goal → goal → goal
val (|||): goal → goal → goal
val call_fresh: (v → goal) → goal
```

Конъюнкция "&&&" объединяет результаты своих целей-аргументов с помощью `bind`, дизъюнкция "|||" конкатенирует результаты с помощью `mplus`, примитив `call_fresh` принимает абстрагированную цель и применяет её к только что созданной свежей переменной. Тип v соответствует типу свежей логической переменной, детали которого мы обсудим позже (раздел 6). Эти комбинаторы являются базовыми для реализации более удобных конструкций реляционного программирования (`conde`, `fresh` и т.д.)

Наконец, существуют два примитива для создания примитивных целей: ограничения унификации и ограничения-неравенства. Тип термов t также оставим абстрактным и вернёмся к нему позже.

```
val (===): t → t → goal
val (=/=): t → t → goal
```

В различных реализациях miniKanren все конструкции можно реализовывать сходным образом. Но в случае с OCaml реализация полиморфной унификации (раздел 5) потребовала использования нетривиальных решений, которые отсутствуют в оригинальном miniKanren.

С использованием объявленных выше типов, сигнатура примитива run выглядит так:

```
val run: goal → state stream
```

Данная функция создает начальное состояние и применяет к нему цель. Состояния в получившемся потоке описывают различные решения данной цели. Так как поток строится постепенно, для проведения поиска следуют получать состояния по одному.

Конкретные ответы реконструируются из состояний. Как правило, реифицируются в состоянии сразу несколько переменных, т.е. оттуда достаются их конкретные значения. Также для свободных переменных извлекаются ограничения-неравенства, представленные как список "запрещённых" термов. Так как эти запрещённые термы могут в свою очередь содержать другие свободные переменные, то реификация ограничений осуществляется рекурсивно.

В случае OCaml реификация является нетривиальной частью реализации, так как, как мы увидим, она не может быть реализована на типобезопасном фрагменте языка.

5. Полиморфная унификация

Довольно естественно попытаться реализовать полиморфную унификацию в языке, оснащенном полиморфным сравнением — удобной, но порою спорной

функциональностью[11]. Как и полиморфное сравнение, полиморфная унификация производит обход значений, используя знания о представлении значений во время выполнения программы. Неоспоримым преимуществом данного подхода является то, что, во-первых, для использования унификации программисту не придется писать стереотипного кода, а во-вторых, то, что данный подход позволяет получить наиболее эффективную реализацию. С другой стороны, недостатки полиморфного сравнения также наследуются, в частности унификация будет зависать на циклических структурах данных и не работать со значениями-функциями. Так как мы не ожидаем никакого разумного поведения в этих случаях, основной проблемой останется то, что компилятор не будет способен предупредить об использовании этих крайних случаев. Другим недостатком является то, что реализацию придется изменять каждый раз, когда представление данных в компиляторе поменяется. Также полиморфная унификация реализована в слабо типизированной манере с использование модуля Obj, и её необходимо аккуратно тестировать.

Важным различием между полиморфным сравнением и унификацией является то, что сравнение только изучает свои операнды, а унификация постепенно конструирует подстановку-результат. Подстановка отображает переменные в термы и используется далее для реификации ответов. Итого, нам необходимо показать, что в результате не конструируются неправильно типизированные значения. Может показаться, что это свойство будет соблюдаться естественным образом, так как сутью синтаксической унификации является проверка того, что некоторые объекты считаются равными. Тем не менее существуют различные системы типов и различные реализации унификации, к тому же иногда *одинаковые значения* могут быть типизированы *по-разному*, поэтому ниже будет представлено обоснование корректности в абстрактном случае, а его части будут переиспользованы для конкретных.

Для начала рассмотрим три алфавита:

- $\tau, \ldots$ — типы;
- $x^{\tau}, \ldots$ — типизированные логические переменные;
- $C_k{}^{\tau}\ (k \geq 0), \ldots$ — конструкторы типа τ.

Множество всех правильно построенных типов задается индуктивно:

$$t_{\tau} = x_{\tau} \mid C_k{}^{\tau}\ (t_{\tau 1}, t_{\tau 2}, \ldots, t_{\tau k}),$$

подразумевая, что для любого конструктора и для всех его вхождений его аргументы имеют соответствующие типы.

Будем считать, что всем термам во время компиляции соответствует некоторый тип. Воспользуется этим свойством чтобы реализовать алгоритм унификации на OCaml, используя некоторые представления для термов и типов:

```
val unify: term → term → subst option → subst option,
```

где “term” соответствует типу представления типизированных термов, а “subst” — это подстановка (частичное отображение переменных в термы). Унификация может закончиться неуспешно (поэтому в типе результат присутствует “option”), выполняется в контексте некоторой подстановки (поэтому “subst” в третьем аргументе) и может запускаться по цепочке (для этого “option” в третьем аргументе).
Используется тот же самый алгоритм унификации с треугольной подстановкой, что и в

[11] Например, https://blogs.janestreet.com/the-perils-of-polymorphic-compare (проверено 10 января 2025)

минимальной реализации [5] miniKanren. Также опущены некоторые не очень важные детали (такие как “occurs check”), но необходимые в настоящей реализации, а также воздержимся от обсуждения алгоритма самого по себе: превосходное описание [21] вместе с доказательством корректности можно найти в других источниках.

Следующий псевдокод демонстрирует схему реализации:

```
let rec unify t^τ1 t^τ2 subst = (* 1 *)
 let rec walk s t^τ =
 match t^τ with
 | x^τ when x^τ ∈ dom(s) -> walk s (s x^τ) (* 4 *)
 | _ -> t^τ
 in
 match subst with
 | None -> None
 | Some s ->
   match walk s t^τ1, walk s t^τ2 with (* 10 *)
   | x^τ1, x^τ2 when x^τ1 = x^τ2 -> subst
   | x^τ1, q^τ2 -> Some (s [x^τ1 ← q^τ2])
   | q^τ1, x^τ2 -> Some (s [x^τ2 ← q^τ1])
   | Cτ (t^τ1 1, …, t^τk k), Cτ (p^τ1 1, …, p^τk k) ->
      unify t^τk k p^τk k(… (unify t^τ1 1 p^τ1 1 subst) …) (* 15 *)
   | _, _ -> None
```

Напоминаем читателю, что в верхних индексах указаны типы, которые мы считаем частью значений. Например, в строке 1 имеется в виду, что мы запускаем unify от термов t^1 и t^2, у которых один и тот же тип τ. Предположим, что самый первый запуск унификации происходит от термов одного типа, и что любая подстановка может быть получена только с помощью одной или нескольких унификаций.

Покажем, что в этих предположения все атрибуты с типами не нужны: они не влияют на исполнение unify и могут быть стёрты. Единственное место, где невозможно предъявить атрибут явно — это строка 4, где с помощью функции walk возвращается результат применения подстановки. Можно показать по индукции, что любая подстановка обладает следующим свойством: если в подстановке s определено значение переменной x^τ, то это значение имеет тип τ, и, следовательно, walk s t^τ всегда возвращает значения типа τ. Очевидно, что это свойство выполняется для пустых подстановок. Рассмотрим произвольную подстановку s, для которой это свойство выполнено. В 10й строке совершаются два вызова — walk s $t^{\tau 1}$ и walk s $t^{\tau 2}$ — и разбираются их результаты. По индукционному предположению у них тип τ и производить сопоставление с образцом в строке 14 можно. В сроке 11 подстановка возвращается без изменений, в строках 12 и 13 — изменяется с сохранением свойства. Наконец, в 15й строке осуществляется несколько вызовов unify, но все они запускаются на термах одинакового типа, а значит, что подстановка обладает интересующим нас свойством. Индукция по структуре терма завершает доказательство.

Итак, атрибуты с типами не важны — они никогда не анализируются и не ограничивают сопоставления с образцом, а значит могут быть полностью стёрты. Также стоит обратить внимание, что представлением термов может служить обычное представление среды исполнения OCaml. Но это нельзя сделать естественным способом, придется

воспользоваться низкоуровневым интерфейсом модуля Obj[12]. Также понадобится некоторый способ отличать вхождения логических переменных в терм, в оригинальной реализации эта проблема решалась с помощью договоренности — роскошь, которую мы не может себе позволить. Мы вернемся к этому вопросу в следующем разделе.
Реализация унификации называется *полиморфной*, потому что на верхнем уровне ей можно приписать тип

```
val unify: 'a → 'a → subst option → subst option
```

Тип подстановки не является полиморфным, что означает, что компилятор полностью теряет типы значений, хранящихся в подстановке. Они восстанавливаются во время реификации ответов (раздел 7). Вне унификации компилятор поддерживает типизацию, что означает что все термы, подтермы и логические переменные согласуются по типам во всех контекстах.

6. Представление термов и их погружение в логический домен

Полиморфная унификация из предыдущего раздела работает для значений произвольных типов, при условии, что можно отличать логические переменные от других термов. Это предположение зависит от вида представления термов. В оригинальной реализации на языке Scheme все термы были обычными s-выражениями, а логические переменные (в простом случае) — одноэлементными векторами. Пользователь обязан выполнять это соглашение и воздерживаться от работы с пользовательскими значениями, состоящими из векторов.

В нашем случае мы хотим сохранить и строгую статическую типизацию и автоматический вывод типов. Так как была выбрана полиморфная унификация, нежелательно представлять логические значения различных типов по-разному (технически это возможно, но будет противоречить выбранному легковесному подходу). Из этого следует, что термы с логическими переменными должны быть типизированы не так, как типизируется термы, объявленные пользователем, иначе будет возможно использовать термы в контекстах, где логические переменные не отображаются должным образом. В то же время нежелательно, чтобы типы логических значений были полностью отделены от пользовательских типов: могут пригодиться конструкторы, объявленные пользователем и т.п. Эти рассуждения подводят к идее логического представления для пользовательских типов данных. Неформально, логическим представлением типа t является тип ρ_t, для которого определена пара функций:

$$\uparrow: t \to \rho_t - \text{инъекция}$$

$$\downarrow: \rho_t \to t - \text{проекция}.$$

Тип ρ_t структурно повторяет t, но может содержать логические переменные. Также инъекция — это тотальная функция, а проекция — частичная.

Важно спроектировать представление значений следуя некоторой обобщённой схеме, иначе комбинаторы miniKanren будет невозможно правильно протипизировать. Также желательно представить обобщённый способ получения пар инъекции/проекции (а ещё лучше, автоматизировать), чтобы снять с программиста груз ответственности за правильную их

[12] https://ocaml.org/manual/5.3/api/Obj.html (проверено 28 января 2025 г.)

реализацию. Наконец, представление должно позволять надежным образом отличать логические переменные от других значений.

В этом разделе рассмотрен два способа реализовать логические представления. Первый достаточно просто реализовать, но, к сожалению, реализация будет показывать низкую производительность на практически значимых примерах. Чтобы исправить этот недостаток, был разработан другой способ представления значений, который переиспользует некоторые компоненты первого. В разделе 10 представлены результаты экспериментального сравнения производительности.

6.1. Тегированные логические значения

Наиболее естественным решением будет тегированное логическое представление. Вводится полиморфный тип $[\alpha]$[13], который соответствует логическому представлению типа α.

```
type [α] = Var of int | Value of α
```

Неформально выражаясь, любое значение типа $[\alpha]$ является либо значением типа α, либо свободной логической переменной. Обращаем внимание, что конструкторы этого типа не используются пользователем напрямую, так как единственным способом создания переменных является конструкция “fresh”, т.е. тип логических значений является абстрактным. Используя этот тип можно определить сигнатуры примитивов унификации, неравенства и создания переменных следующим образом:

```
val (===): [α] → [α] → goal
val (=/=): [α] → [α] → goal
val call_fresh: ([α] → goal) -> goal
```

И ограничение унификации, и неравенства использует одну и ту же полиморфную унификацию, но их тип ограничен только логическими значениями.

Кроме переменных, другие логические значения также могут быть получены с помощью инъекции. И наоборот, иногда логические значения могут быть спроецированы в обычные. Для этого предоставляются два базовых примитива[14], которые могут быть использованы для *поверхностной* инъекции/проекции. Как и ожидается, инъекция тотальная, а проекция — частичная функция.

```
val (↑∀): α → [α]
val (↓∀): [α] → α
let (↑∀) x = Value x
let (↓∀) = function Value x → x | _ → failwith "not a value"
```

Пара поверхностных функций хорошо работает для примитивных типов, но чтобы реализовать инъекцию/проекцию для произвольных, необходимо воспользоваться идеей представления типов в виде неподвижных точек функторов [15]. Для наших нужд желательно сделать функторы полностью полиморфными: такими типами, где логические значения можно писать в произвольных позициях. К тому же данный подход позволит переиспользовать имеющийся код с меньшим количеством изменений.

Продемонстрируем этот подход на каноническом примере связных списков. Рассмотрим обычный тип данных OCaml для полиморфных списков:

[13] В конкретном синтаксисе “α logic”.
[14] В конкретном синтаксисе это “inj” and “prj”.

```
type α list = Nil | Cons of α * α list
```

В этом типе можно использовать логические переменные только на месте элементов списка, но не вместо “хвоста”, так как там ожидается значение типа α list, зафиксированного в конструкторе Cons. Чтобы получить подходящее логическое представление, вначале абстрагируем тип полностью полиморфного функтора:

```
type (α, β) L = Nil | Cons of α * β.
```

Таким образом, оригинальный тип может быть выражен как

```
type α list = (α, α list) L,
```

а его логическое представление как

```
type α listº = [([α], α listº) L]
```

Более того, с помощью специфичной для функтора функции fmap

```
val fmapL: (α → α') → (β → β') → (α, β) L → (α', β') L
```

можно реализовать инъекцию и проекцию:

```
let rec ↑list l = ↑∀(fmapL (↑∀) ↑list l)
let rec ↓list l = fmapL (↓∀) ↓list (↓∀ l)
```

Специфичные для функторов функции fmap могут быть легко написаны, и даже получены автоматически с помощью различных методов обобщённого программирования для OCaml. А с помощью fmap уже будут написаны инъекции и проекции пользовательских типов данных.

Теперь вернемся к вопросу определения переменных во время полиморфной унификации. Так как типы не известны, мы не можем отличать переменные с помощью простого сопоставления с образцом. В реализации переменные проверяются следующим способом:

- в окружении хранится дополнительное значение-якорь, создаваемое вместе с начальным состоянием; этот “якорь” не изменяется для всех производных состояний во время запуска run;
- в тип логических переменных добавлен “якорь”, который при берется из окружения и при создании переменных кладется в них.
  ```
  type [α] = Var of int * anchor | Value of α
  ```
- во время унификации, для проверки переменных проверяется конъюнкция следующих свойств:
  - рассматриваемое значение не является примитивным (т.е. числом);
  - тег и раскладка в памяти соответствует переменным, т.е. значениям, построенным с помощью конструктора Var из типа [α];
  - адрес якоря в памяти соответствует адресу якоря в текущем окружении.

Учитывая, что состояние абстрактно для пользователя, можно гарантировать, что только переменные, созданные в текущем запуске примитива run, пройдут тест на переменные, так как указатель на якорь уникален среди всех указателей и никак может попасть во вне из-за использования примитива создания переменных.

Осталось описать фазу реификации, которая может быть представлена с помощью следующей функции:

```
val reify: state -> [α] -> [α]
```

Функция принимает состояние и логическое значение, и рекурсивно совершает подстановку всех переменных в логическом значении до тех пор, пока остаются связанные переменные. Так как наша реализация подстановки не является полиморфной, то функция reify также реализована в небезопасном стиле. Но легко показать, что reify не создает неправильно

типизированных термов. Все термы в подстановке корректно типизированы, подстановка осуществляет замену переменных на термы такого же типа, а унификация не изменяет типы унифицируемых термов. Следовательно, reify всегда подставляет в правильно типизированном терме подтермы на термы таких же типов, что обуславливает безопасность системы типов.

Кроме осуществления подстановки функция reify также реифицирует ограничения-неравенства. При этом к каждой свободной переменной приписывается список реифицированных термов, представляющий ограничения-неравенства для этой свободной переменной. Заметим, что ограничения неравенства могут быть созданы только для одинаково типизированных термов, что обосновывает типовую безопасность реификации. Обратим также внимание, что нужны дополнительные проверки для избежания зацикливания реификации, так как реификация ответов и ограничений взаимно рекурсивны, а реификация ответа может вызвать саму себя по цепочке ограничений-неравенств. В примере ниже ответ для переменной q будет содержать неравенства для переменной r; реификация r приведет ограничения с переменной s, которая снова запустит реификацию переменной r, и т.д.

```
let foo q =
  fresh (r s)
    (q === ↑∀ (Some r)) &&&
    (r =/= s) &&&
    (s =/= r)
```

6.1. Логические переменные без тегирования. Отслеживание типов

Решение, предложенное выше, страдает существенным недостатком: чтобы произвести унификацию, необходимо погрузить термы в логический домен, что сделает их в два раза больше. В результате, эта реализация проиграет по производительности оригинальной.

Было разработано другое представление без этого недостатка. Идея в том, чтобы хранить данные (не переменные) в памяти без использования тегирования вообще.

```
type [α] = α
```

Как следствие станет невозможно конструировать данные с переменными внутри путём использования конструкторов, а придется использовать специальные функции-конструкторы. В нашей реализации тип [α] абстрактный, поэтому эти модификации не меняют интерфейса взаимодействия с данными из реляционного кода. Но при этом всё ещё возможно представлять переменные старым способом, переиспользовав алгоритм проверки на переменные, а реализация полиморфной унификации остается *почти* такой же. Некоторой проблемой является то, что можно вводить переменные в нелогические и нетегированные данные.

Эти свободные переменные не помещают унификации (она сумеет их отличить), но их нельзя оставить для нелогического домена как они есть.

Идея заключается в том, чтобы использовать в общем случае некорректное представление в памяти — ему нельзя придать тип в языке OCaml — для внутренних нужд, а применять тегирование только на фазе реификации. Но тогда необходимо решить каким будет тип функции reify. Прямолинейное решение `val reify: state -> [α] -> [α]` не годится, так как тип аргумента совпадает с типом результата [α]. Хотелось бы что-то похожее на

```
val reify: state -> [α] -> («tagged» [α]).
```

Если α — тип без параметров, то можно затегировать результат конструкторами Var

или Value в зависимости того переменная это или нет. Всё несколько сложнее для типов с параметрами, например, для типа целочисленных списков [[int] list]. Здесь надо не только тегировать не верхнем уровне, но и запускать реификацию с тегированием рекурсивно для подвыражений внутри. Итого, необходим следующий (мета)тип для функции реификации, где β — тип затегированного α:

```
val reify: state -> [α] -> («tagged»[β]).
```

Эти наблюдения обосновывают выбранную конкретную реализацию.

Оригинальное определение типа [α] будет тегированным представлением. Введем дополнительный однопараметрический тип "α ilogic", представляющий логические значения типа α.

```
type α ilogic = α
val inj: α -> α ilogic
```

Для пользователя он будет абстрактным, а его значения будут появляться с помощью функции поверхностной инъекции inj. Для рекурсивных типов всё несколько сложнее: необходимо применять инъекцию рекурсивно для всех подвыражений. Рассмотрим эту инъекцию на примере списков.

```
type α list = (α, α list) L,
type α injected = (α, α injected) L ilogic,
type α listo = [([α], α listo) L]
```

В типе нетегированного представления необходимо расставить тип ilogic для всех подвыражений, так образом инъекция целочисленных списков с типом int list будет порождать значения типа `int ilogic injected`. На практике для построения логических представлений списков стоит использовать специальные функции-конструкторы вместо конструкторов алгебраического типа.

```
let cons h tl: α -> α injected -> α injected =
        inj (Cons (h, tl))
let nil (): unit -> α injected = inj Nil
```

Также необходимо реализовать реификацию значений из логического домена. На данный момент подход к реификации не ограничивает тип, куда реификация будет производится, но каноническим является использование упомянутого выше [α]. Все реификаторы являются двухпараметрическими типами, инкапсулирущими преобразование из логического домена в некоторый кодомен. Например, для списков необходимо получить реификатор с типом

```
(int ilogic injected, [int] listo) reifier.
```

Ниже пример реификатора для списков, который можно построить, предъявив реификатор `ra` для элементов списка. Сами реификаторы по сути являются монадами Reader, которые хранят в себе информацию для определения переменных в терме. Типы реификаторов абстрактные, чтобы пользователь не смог провести реификацию переменных, созданных не в том состоянии. При реализации реификации используется интерфейс аппликативных функторов с операциями `pure` и `<*>`. Также используются комбинаторы неподвижных точек для реификаторов (`Reifier.fix`) и для call-by-value языков (`Reifier.zed`), а ещё специальная функция `chain`. Примитив `reify` проводит поверхностную реификацию, а функция `rework` рекурсивно запускает реификацию для подвыражений и ограничений-неравенств, если таковые присутствуют.

```
val chain: (α reifier -> β reifier) -> (α -> β) reifier
let fmapt f g xs = pure (fmap_list) <*> f <*> g <*> xs
let reify: (α, β) reifier -> (α injected, β list^o) reifier =
fun ra ->
  Reifier.fix (fun self ->
    OCanren.reify <..> chain
      (Reifier.zed
        (Reifier.rework ~fv:(fmapt ra self)) ) )
```

Функции `chain`, `OCanren.reify` и комбинаторы неподвижной точки определены один раз для всех типов, и пользователю для своего типа данных необходимо реализовать только `fmapt`, `reify` и функции-конструкторы, упомянутые выше. Для облегчения этого было разработано синтаксическое расширение, которое по объявлению типа порождает всё необходимое во время компиляции. Пример его использования можно найти в разделе 9.

7. Реификация и примитивы верхнего уровня

В разделе 4 был упомянут примитив верхнего уровня `run`, который позволяет запускать цель и возвращать поток состояний. Чтобы получить ответы на запрос, представленный целью, все переменные должны быть реифицированы в соответствующих состояниях, с использованием примитивов реификации из раздела 6. Но состояния содержат ответы в нетипизированном представлении, и типы ответов восстанавливаются только на основе типов реифицируемых переменных. Таким образом, корректность типов после реификации критически зависит от требования, чтобы переменные реифицировались только в тех состояниях, которые являются потомками (в смысле дерева поиска) того состояния, где эти переменные были созданы. В этом разделе описан набор примитивов, который помогает поддерживать это требование.

Ниже представлен набор комбинаторов, которые вызывают реляционный код и проводят реификацию только в правильных состояниях. Примитив верхнего уровня `run` реализован заново, и теперь принимает несколько аргументов. Его конкретный тип довольно длинен, и поэтому описан пример использования этого комбинатора:

```
run n (fun l1 ... ln -> G) (fun a1 ... an -> H).
```

Здесь `n` является нумералом, который описывает количество аргументов у других параметров `run`, $l_1 \ldots l_n$ — это логические переменные; G — цель, которая может использовать упомянутые логические переменные $l_1 \ldots l_n$; $a_1 \ldots a_n$ — реифицированные ответы для переменных $l_1 \ldots l_n$ соответственно; и, наконец, `H` — обработчик, который может использовать $a_1 \ldots a_n$; новый `run` возвращает ленивый поток результатов вычисления обработчиков.

Типы $l_1 \ldots l_n$ соответствуют логическому домену и всегда состоят из применения на верхнем уровне конструктора типов ilogic. Типы $a_1 \ldots a_n$ однозначно определяются типами $l_1 \ldots l_n$, и по сути являются функциями, которые по реификатору восстанавливают реифицированное значение соответствующего типа для нужной переменной. Как правило, реификаторы бывают двух видов: описанный в предыдущем разделе `reify` и его упрощенный аналог, который аварийно завершается при наличии свободных переменных, но если их нет, то порождает значение, в типе которого свободных переменных не предусмотрено. Последний может быть полезен для случаев, когда программист уверен в своей реляционной программе и хочет упростить типы для более легкой интеграции с нереляционным кодом.

Несколько нумералов (с названиями q, qr, qrs) для 1, 2 и 3-аргументных отношений объявлены заранее, остальные получаются с помощью применения функции следующего нумерала, которая принимает нумерал и увеличивают количество ожидаемых аргументов на единицу. Реализация этой функции сделана на основе работ [9, 10] о поливариадических функциях.

Итого, поиск и реификация тесно связаны: невозможно выполнить реификацию в произвольном состоянии для произвольной переменной. Такое решение гарантирует корректность типов и освобождает программиста от проведения реификации полностью вручную.

8. Ограничения-неравенства

Зачастую для конкретной задачи пользователи расширяют реализацию реляционного программирования новыми ограничениями. В этом разделе мы расскажем про ограничение неравенства, которое, строго говоря не обязательно, но полезно для широкого круга задач.

8.1. Задача удаления элемента из списка

Напишем отношение, которое принимает список и некоторый элемент, и возвращает список без первого вхождения этого элемента.

```
let rec rember°1 x xs out =
  conde
    [ xs === Std.nil () &&& (xs === out)
    ; fresh (h tl) (xs === Std.List.cons h tl) (h === x)
                   (tl === out)
    ; fresh (h tl res)
      (xs === Std.List.cons h tl)
      (out === Std.List.cons h res)
      (rembero1 x tl res)]
```

Реализация повторяет оригинальную [2] для Scheme. Случай пустых списков тривиален, иначе нам надо либо удалить головной элемент h, если он равен искомому x, либо проигнорировать головной элемент и запуститься рекурсивно для хвоста списка.

Приведенная реализация успешно удаляет первое вхождение элемента и выдает ожидаемый ответ первым. Но если запросить с помощью `run` больше ответов, то, неожиданно, они найдутся. Дело в том, что реляционный поиск попробует воспользоваться третьей ветвью поиска даже если искомый элемент находится в голове, и поэтому реляционная программа выше вернет по ответу на каждое удаление элемента x, а также, не удалив ни одного x, ещё один ответ — исходный список.

```
run * (fun q -> rember°1 !!2 [ 1; 2; 3; 2; 4] q)
-> q=[1; 3; 2; 4]
-> q=[1; 2; 3; 4]
-> q=[1; 2; 3; 2; 4]
```

```
let rec rember°2 x xs out =
   conde
      [ xs === Std.nil () &&& (xs === out)
      ; fresh (h tl) (xs === Std.List.cons h tl)
                     (h === x) (tl === out)
      ; fresh (h tl res)
          (xs === Std.List.cons h tl)
          (h =/= x)
          (out === Std.List.cons h res)
          (rembero2 x tl res)]
```

Чтобы решить эту проблему, необходимо сделать вторую и третью ветви conde несовместными, сказав явно в третьей ветви, что x не может быть равен h. После этой модификации реляционная программа `rember°2` перестанет давать неожиданные ответы.

```
let rec rember°2 x xs out =
  conde
    [ xs === Std.nil () &&& (xs === out)
    ; fresh (h tl) (xs === Std.List.cons h tl)
                   (h === x) (tl === out)
    ; fresh (h tl res)
        (xs === Std.List.cons h tl)
        (h =/= x)
        (out === Std.List.cons h res)
        (rembero2 x tl res) ]
```

8.2. Реализация проверки ограничений-неравенств

Самым простым способом реализации ограничений-неравенств является их проверка с помощью унификации. Для каждого ограничения из двух термов мы проводим их унификацию и действуем в зависимости от результата.

1. Если их унифицировать невозможно, то термы никогда не будут равными и ограничение можно выкинуть.
2. Если они унифицируются без расширения текущей подстановки, то термы являются равными в этой ветви поиска. Ограничение нарушено, поиск стоит продолжать в других ветвях.
3. Если термы унифицируются с путём добавления некоторых пар переменная-терм в текущую подстановку («дельта»), то ограничение является содержательным, и его надо повторно проверять позже.

К данной базовой реализации можно применить следующую оптимизацию, реализованную в OCanren и faster-miniKanren. Заметим, что для полученной дельты если хотя бы одна пара переменная-терм унифицируется с непустой подстановкой, то скорее всего оригинальные два терма унифицируются с непустой подстановкой. Таким образом, можно сократить размер термов, которые надо унифицировать для проверки ограничений, что приводит к улучшению производительности.

8.3. Ограничения-неравенства и сложные типы данных

Неаккуратное использование ограничений-неравенств для сложных типов данных может приводить к неожиданным результатам. Рассмотрим реляционную спецификацию, которая постулирует, что переменная q является натуральным числом (заданным в стиле Пеано) минимум с тремя конструкторами s (следующий). Другими словами, $q \geq 3$.

```
fun q -> fresh (tl) (q === s (s (s tl)))
```

При попытке выразить обратное (число меньше трех), заменив унификацию на ограничение неравенства, мы не добьёмся ожидаемого результата, так как для любого числа > 3 найдется такое значение логической переменной q, что ограничение будет выполненным, т.е. его можно просто выкинуть, оно не влияет на результат. Ожидаемый результат получился бы, если бы конструкция fresh здесь ввела бы универсально квантифицированную переменную, но она вводит экзистенциальную. Добавление универсальной квантификации в miniKanren исследовалось[15], но на данный момент эффективная и корректная реализация не найдена.

Для решения этой проблемы в OCanren добавлен новый вид переменных — шаблонные (англ. wildcard) логические переменные, которые обозначаются в синтаксисе с помощью двух подчеркиваний. Их можно использовать на месте любого логического значения, но в реализации все шаблонные переменные представлены одинаково. Они так названы в честь одноименной конструкции языка OCaml, которая позволяет при сопоставлении с образцом выразить, что пользователю не важно, как устроена часть терма. Во время проверки ограничений-неравенств шаблонные переменные обрабатываются максимально пессимистичным образом: вместо них подразумевается терм, похожий на то, с чем унифицируется соответствующая шаблонная переменная.

Рассмотрим несколько примеров. Неравенство пар (1, _.10) и (__, 2) упрощается до неравенства переменной 10 и числа 2. Вместо шаблонной переменной пессимистично подставляется число 1, делая первые компоненты пар одинаковыми. Неравенство двух шаблонных переменных также упрощается максимально пессимистично: вместо переменных подставляется одинаковое значение и ограничение тривиально нарушается. Проверка неравенства свежих переменных и термов с шаблонными переменными внутри откладывается до момента уточнения свежих переменных. Например, ограничение (cons __ __ =/= _.10) после уточнения 10й переменной с помощью унификации (cons _.11 _.12 === _.10) упростится до “либо 11я, либо 12я переменная не унифицируется с шаблонной”. Разрешение неравенств свежих и шаблонных переменных откладывается на потом, до момента уточнения значений свежих переменных с помощью унификации.

Проверка неравенства шаблонных переменных и частично заданных термов заканчивается пессимистично и неуспешно. Именно поэтому пример со числами Пеано выше показывает ожидаемое поведение: достаточно большие числа “n” сведутся к проверке неравенства шаблонной переменной и “n-3”, для малых унификация закончится с ошибкой, не доходя до шаблонной переменной. Данный пример можно переформулировать для других алгебраических типов данных, погруженных в реляционный домен. В контексте оригинальной реализации на языке Scheme, шаблонные переменные также применимы, но с некоторыми оговорками. Например, встроенные в язык списки позволяют естественным образом представлять “списки длины хотя бы три”, поэтому в Scheme стоит демонстрировать шаблонные переменные с числами Пеано.

Шаблонные переменные применимы не только на “игрушечном” примере, показанном выше. Автоматический синтез [22] реляционных программ из функциональных может воспользоваться шаблонными переменными при обработке сопоставления с образцом. На

[15]Ограничение eigen: https://github.com/webyrd/miniKanren-with-symbolic-constraints (проверено 25 июня 2025).

данные момент ветви сопоставления с образцом отображаются в набор реляционных спецификаций, соединенных дизъюнкцией с помощью conde (рис. 2a). Это приводит к неожиданным результатам в виде дополнительных ответов, так как ветки conde независимы, а при сопоставлении с образцом выбирается всегда одна. В этом случае шаблонные переменные позволяют описать (рис. 2b) в спецификации набор ограничений,

```
match x, y, z with
| _, F, T -> 1
| F, T, _ -> 2
| _, _, F -> 3
| _, _, T -> 4
```

Рис. 1. Пример сопоставления с образцом в языке OCaml.
Fig. 1. An example of pattern matching in OCaml.

```
let enc_with_diseq q rez =
let _T = !!true in
let _F = !!false in
let w = Std.triple in
conde
[ fresh (fresh1)
    (rez === !!1)
    (q === w fresh1 _F _T)
; fresh (fresh1 x)
   (rez === !!2)
   (q === w _F _T fresh1)
   (q =/= w x _F _T)
; fresh (fr1 fr2 x y z)
  (rez === !!3)
  (q === w fr1 fr2 _F)
  (q =/= w x _F _T)
  (q =/= w _F _T z)
; fresh (fr1 fr2 x y z)
  (rez === !!4)
  (q =/= w x _F _T)
  (q =/= w _F _T z)
  (q =/= w x y _F)
  (q === w fr1 fr2 _T)
]
```

```
let enc_with_wc q rez =
let _T = !!true in
let _F = !!false in
let w = Std.triple in
conde
[ fresh ()
   (rez === !!1)
   (q === w __ _F _T)
; fresh ()
   (rez === !!2)
   (q === w _F _T __)
   (q =/= w __ _F _T)
; fresh ()
   (rez === !!3)
   (q === w __ __ _F)
   (q =/= w __ _F _T)
   (q =/= w _F _T __)
; fresh ()
    (rez === !!4)
    (q =/= w __ _F _T)
    (q =/= w _F _T __)
    (q =/= w __ __ _F)
    (q === w __ __ _T)
]
```

Рис. 2. Две возможные трансляции примера 1. Слева с ограничениями-неравествами. Справа с шаблонными переменными.
Fig. 2. Two possible translation of Fig. 1. Disequality constraints on the left. Wildcard variables are on the right.

9. Примеры

В этом разделе представлены несколько примеров реляционных программ, реализованных с помощью описываемого подхода. Кроме комбинаторов miniKanren использованы синтаксические расширения: одно вводит конструкцию fresh, делая код более похожим на оригинальный miniKanren, другое необходимо для порождения необходимых функций-конструкторов и реификаторов. Также используется небольшая библиотека структур данных, полезных при написании реляционных программ: списки, числа Пеано, булевы значения и т.п. Библиотека реализована полностью на стороне пользователя, методом из раздела 6, без использования небезопасных конструкций компилятора OCaml.

9.1. Конкатенация и обращение списков

Введение в реляционное программирование обычно излагается с помощью отношений конкатенации и переворота списков, и мы не будет отступать от этой традиции. Реализация отношения append o на оригинально miniKanren было приведено ранее в разделе 3. В случае OCanren реализация выглядит похоже:

```
let rec append° x y xy = conde [
  (x === List.nil ()) &&& (y === xy);
  (fresh (h t ty)
    (x === h % t)
    (h % ty === xy)
     (append° t y ty))]

let rec revers° a b =
  conde [
   (a === List.nil ()) &&& (b === List.nil ());
   (fresh (h t a')
     (a === h % t)
     (append° a' !<h b)
     (revers° t a'))]
```

Выше использовалась реализация реляционных списков, которая предоставляет сокращения для функций-конструкторов:

- “`List.nil ()`” соответствует пустому списку “`[]`”;
- “`h % t`” — “`h::t`”;
- “`a %< (b %< (c !< d))`” — “`[a; b; c; d]`”.

В нашей реализации базовый примитив miniKanren “conde” реализован как дизъюнкция целей, а не как макрос. Также у нас доступны явные конъюнкции и дизъюнкции как инфиксные операторы. В оригинальной реализации для Scheme использовались вложенные скобки, но мы считаем, что для OCaml это слишком неестественно.

9.2. Реляционная сортировка и перестановки

В следующем примере рассматривается сортировка списков. В качестве элементов используются натуральные числа в виде Пеано, потому что библиотека OCanren их поддерживает, но любые другие типы данных, оснащённые отношением линейного порядка, тоже подойдут.

Сортировку чисел на miniKanren можно реализовывать по-разному, потому что любой алгоритм сортировки списков может быть переписан реляционно. Ниже же будем реализовывать максимально декларативно, чтобы продемонстрировать преимущества реляционного подхода. Будем утверждать, что все пустые списки являются сортированными, и что сортированный непустой список является конкатенацией минимального элемента с сортированным списком из всех оставшихся элементов. Реляционная спецификация ниже реализует сортировку по определению выше:

```
let rec sort o x y = conde
  [ (x === List.nil ()) &&& (y === List.nil ())
  ; fresh (s xs xs')
      (y === s % xs')
      (sorto xs xs')
      (smallesto x s xs)]
```

Выражение «`smallest° x s xs`» означает, что «s» является наименьшим элементов списка «xs», а в «xs» хранятся все оставшиеся элементы. Отношение «`smallest°`» можно реализовать путём разбора случаев.

```
let rec smallest° l s l' =
  conde [
    (l === s % nil ()) &&& (l' === nil ());
    fresh (h t s' t' max)
     (l' === max % t')
     (l === h % t)
     (minmax o h s' s max)
     (smallest° t s' t')
]
```

Также необходимо реализовать отношение для минимума и максимума, используя встроенные в OCanren отношения «`leo`» и «`gto`» для сравнения чисел Пеано:

```
let minmaxo a b min max =
  conde [
     (min === a) &&& (max === b) &&& (leo a b);
     (max === a) &&& (min === b) &&& (gto a b) ]
```

С помощью реляционной сортировки можно реализовать обычную сортировку целых чисел:

```
let sort l =
  run q (sort° (inj_nat_list l))
    (fun q -> q#reify from_nat_list_reifier)
  |> Stream.take
```

Функция `Stream.take` преобразует ленивую последовательность ответов в список, `inj_nat_list` — это инъекция списка целых в логический список натуральных, `from_nat_list_reifer` — проекция обратно.

За счет реляционного программирования можно воспользоваться тем же отношением sort°, чтобы посчитать все перестановки данного списка. Каждая перестановка сортируется в

один и тот же список. Поэтому задача поиска всех перестановок может быть реляционно переформулирована как поиск всех списков, сортирующихся в данный:

```
let perm l =
  run q (fun q -> fresh (r)
    (sorto (inj_nat_list l) r)
    (sorto q r))
    (fun rez -> rez#reify from_nat_list)
  |> Stream.take ~n:(fact (length l))
```

Заметим, что сортировку списка мы сделали упомянутым выше способом. Также мы запросили ровно факториал длины ответов, потому что большие значения могут привести к расхождению программы.

9.3. Вывод типов для STLC

В качестве последнего примера продемонстрируем вывод типов в просто типизированном лямбда-исчислении (STLC). Решение задачи достаточно просто, чтобы попасть в учебники [1,2], но в нём будет продемонстрировано использование синтаксических расширений для получения инъекций из раздела 6. В качестве подхода к обобщенному программированию у нас используется библиотека GT [23], но любая другая тоже подойдет.

Сначала опишем тип для представления λ-выражений:

```
module Term: sig
  type nonrec ('varname, 'self) t =
  | V of 'varname
  | App of 'self * 'self
  | Abs of 'varname * 'self
  ...
  type 'v injected = ('v, 'v injected) t ilogic
  val v: 'v -> 'v injected
  val app: 'v injected -> 'v injected -> 'v injected
  val abs: 'v -> 'v injected -> 'v injected
end = struct
  [@@ocanren_inject
  type nonrec ('varname, 'self) t =
  | V of 'varname
  | App of 'self * 'self
  | Abs of 'varname * 'self [@@deriving show, fmap]]
end
```

Эту же работу надо повторить для объявлений типов:

```
module Type: sig
  type nonrec ('a, 'b) t =
    | P of 'a
    | Arr of 'b * 'b
  ...
  type 'a injected = ('a, 'a injected) t ilogic
  val o: 'a -> 'a injected
  val arr: 'a injected -> 'a injected
end = struct
  [@@ocanren_inject
  type nonrec ('a, 'b) t =
  | P of 'a
  | Arr of 'b * 'b [@@deriving show, fmap]]
end
```

Реляционная проверка типов напоминает оригинальную реализацию на Scheme: кроме немного другого синтаксиса, вместо конструкторов типов данных используются их логические аналоги:

```
let rec lookup° a g t =
  fresh (a' t' tl)
   (g === (inj_pair a' t') % tl)
   (conde
      [ (a' === a) &&& (t' === t);
        (a' =/= a) &&& (lookup° a tl t) ])

let infer° expr typ =
  let rec infer° gamma expr typ =
    conde [
      fresh (x) (expr === v x) (lookup° x gamma typ);
      fresh (m n t)
        (expr === app m n)
        (infer° gamma m (arr t typ))
        (infer° gamma n t);
      fresh (x l t t')
        (expr === abs x l)
        (typ === arr t t')
        (infer° ((inj_pair x t) % gamma) l t')
      ]
  in
  infer° (List.nil ()) expr typ
```

10. ОЦЕНКА ПРОИЗВОДИТЕЛЬНОСТИ

Изначально, с помощью тестирования производительности мы хотели оценить, какое влияние оказывает выбор языка OCaml для реляционного программирования. В дополнении к этому, реляционное программирование было реализовано эффективно с помощью отказа от тегирования, и влияние этого решения тоже должно быть оценено. Для сравнения был выбрана реализация faster-miniKanren17 для Scheme/Racket. В OCanren были реализованы оптимизации [2], которые применяются в faster-miniKanren. Также была проведена работа по обеспечению обхода дерева поиска одинаковым образом.

Для набора тестов были выбраны следующие задачи:

- **pow, logo** — реляционное возведение в степень и логарифмирование [24] чисел в двоичном представлении. Конкретно выбраны $3^5 = 243$ и $\log_3 243 = 5$.
- **quines, twines, trines** — программы, вычисляющиеся в себя, использованные для тестирования реляционных интерпретаторов [7]. Конкретно тесты вычисляю первые 100, 15 и 2 ответа(ов) для соответствующих задач.

Вычисления производились на компьютере с процессором Intel Core i7-4790K CPU @ 4.00GHz и 16GB памяти. Для OCanren использовался компилятор OCaml-4.14.2+flambda, для faster-miniKanren — Chez Scheme 9.5.8. Все бенчмарки компилировались в нативный код и вычислялось среднее время за 40 испытаний. Результаты представлены на рисунке 3.

Репозиторий с кодом и инструкции по воспроизведению эксперимента доступны на GitHub[16].

Наблюдения показывают, что важность нетегированного представления варьируется от задачи. В зависимости от видов термов, участвующих в унификации, прирост производительности будет разниться. Действительно, если термы являются копиями друг друга, то унификации нужно совершить полный обход термов. Если же термы не равны, то унификация может завершиться раньше, без полного обхода термов.

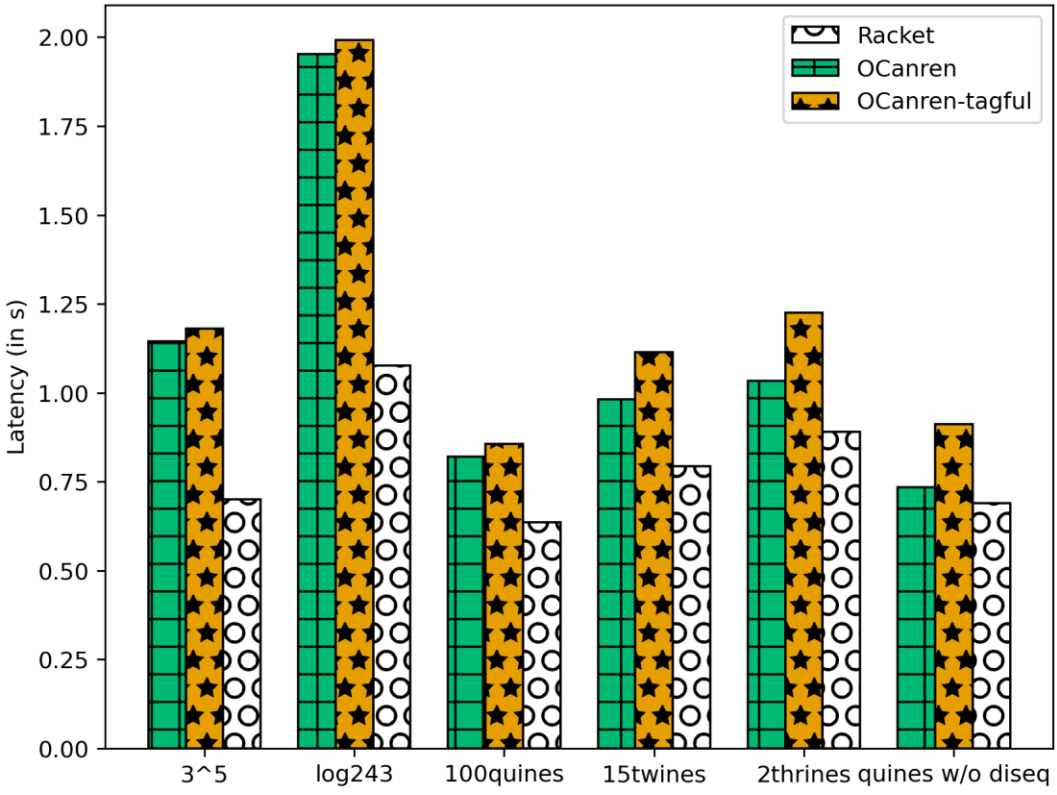

Рис. 3. Результаты оценки производительности
Pic. 3. Benchmark results

Также стоит отметить, что реализация на OCaml отстаёт по сравнению с реализацией для Chez Scheme. Вероятно, это связано с особенностями хранения данных [25] в реализации компилятора. Детальное исследование различий сред исполнения OCaml и Scheme оставлено на будущее.

8. Заключение

Была представлена строго типизированная реализация miniKanren для OCaml. Она проходит все тесты, сделанные для miniKanren, включая ограничения-неравенства, а также реализованы многие интересные программы, известные из литературы. Утверждается, что реализация может быть использована и как удобный реляционный предметно-ориентированный язык, и как платформа для исследования реляционного программирования в целом.

[16] https://github.com/Kakadu/ocanren-perf/tree/ispras-paper2025 (проверено 2 апреля 2025)

Список литературы / References

[1]. Friedman D.P., Byrd W.E., Kiselyov O., Hemann J. The Reasoned Schemer. The MIT Press, 2018.

[2]. Byrd W.E. Relational Programming in MiniKanren: Techniques, Applications, and Implementations. PhD thesis, Indianapolis, IN, USA, 2009.

[3]. Somogyi Z., Henderson F., Conway T. The execution algorithm of mercury, an efficient purely declarative logic programming language. The Journal of Logic Programming, 29(1): 17–64, 1996. High-Performance Implementations of Logic Programming Systems. DOI: 10.1016/S0743106696000684

[4]. Hanus M., H. Kuchen H., and Moreno-Navarro J.J.. Curry: A truly functional logic language. In Proc. ILPS'95 Workshop on Visions for the Future of Logic Programming, pages 95–107, 1995. DOI: 10.1.1.33.7348

[5]. Hemann J. and Friedman D.P. μKanren: A minimal core for relational programming. In Proceedings of the 2013 Workshop on Scheme and Functional Programming (Scheme'13), Month 2013.

[6]. Hemann J., Friedman D.P., Byrd W.E., Might M. A small embedding of logic programming with a simple complete search. In Proceedings of the 12th Symposium on Dynamic Languages, DLS 2016, pages 96–107, New York, NY, USA, 2016. ACM.

[7]. Byrd W.E., Holk E., Friedman D.P. miniKanren, live and untagged: Quine generation via relational interpreters (programming pearl). In Proceedings of the 2012 Annual Workshop on Scheme and Functional Programming, Scheme '12, pages 8–29, New York, NY, USA, 2012. ACM. DOI: https://doi.org/10.1145/2661103.2661105

[8]. Claire E. Alvis, Willcock J.J., Carter K.M., Byrd W.E., Friedman D.P.. cKanren: miniKanren with constraints. In Proceedings of the 2011 Workshop on Scheme and Functional Programming (Scheme '11)., Month 2011. DOI: 10.1.1.231.3635

[9]. Danvy O. Functional unparsing. Journal of Functional Programming, 8(6), 1998. DOI: 10.1017/S0956796898003104

[10]. Fridlender D., Indrika M. Do we need dependent types? Journal of Functional Programming, 10(4), 2000. DOI: 10.1017/S0956796800003658

[11]. Spivey M., Seres S. Embedding prolog in Haskell. In Proceedings of the 1999 Haskell Workshop, 1999. DOI: 10.1.1.35.8710

[12]. Tim Sheard and Jones S.P.. Template meta-programming for Haskell. In Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell, Haskell '02, pages 1–16, New York, NY, USA, 2002. ACM. DOI: 10.1145/581690.581691

[13]. Claessen Koen and Peter Ljungl of. Typed logical variables in haskell. In Electronic Notices in Theoretical Computer Science, volume 41, 200. DOI: 10.1016/S1571066105805444

[14]. Manuel M. T. Chakravarty, Gabriel C. Ditu, and Roman Leshchinskiy. Instant generics: Fast and easy. 2009.

[15]. Swierstra W. Data types á la carte. Journal of Functional Programming, 18(4), 2008. DOI: 10.1017/S0956796808006758

[16]. Jeremy Gibbons. Datatype-generic programming. In Proceedings of the 2006 International Conference on Datatype-generic Programming, SSDGP'06, pages 1–71, Berlin, Heidelberg, 2007. Springer-Verlag. DOI: 10.5555/1782894.1782895

[17]. Yallop J.. Practical generic programming in ocaml. In Proceedings of the 2007 Workshop on Workshop on ML, ML'07, pages 83–94, New York, NY, USA, 2007. ACM. DOI: 10.1145/1292535.1292548

[18]. White L., Bour F., and Yallop J. Modular implicits. In Proceedings ML Family/OCaml Users and Developers workshops, volume 198, pages 22–63, 2015. DOI: 10.4204/EPTCS.198.2

[19]. Baader F., Snyder W. Handbook of Automated Reasoning. Elsevier and MIT Press, 2001.

[20]. Kiselyov O., Shan C., Friedman D.P., and Amr Sabry. Backtracking, interleaving, and terminating monad transformers: (functional pearl). SIGPLAN Not., 40(9):192–203, September 2005. DOI: 10.1145/1090189.1086390.

[21]. Kumar R. Mechanising aspects of miniKanren in HOL. Bachelor Thesis, The Australian National University, 2010.

[22]. Lozov P., Vyatkin A., and Boulytchev D. Typed relational conversion. In Meng Wang and Scott Owens, editors, Trends in Functional Programming, pages 39–58, Cham, 2018. Springer International Publishing. DOI: 10.1007/978-3-319-89719-6_3

[23]. Kosarev D., Boulytchev D. Generic programming with combinators and objects. Scientific and Technical Journal of Information Technologies, Mechanics and Optics, 2021. DOI: 10.17586/2226-1494-2021-21-5-720-726

[24]. Kiselyov O., Byrd W.E., Friedman D.P., Shan C. Pure, declarative, and constructive arithmetic relations (declarative pearl). In Proceedings of the 9th International Conference on Functional and Logic Programming, FLOPS'08, pages 64–80, Berlin, Heidelberg, 2008. Springer-Verlag. DOI: 10.1007/978-3-540-78969-7_7

[25]. Dybvig R.K., Eby D., Bruggeman C. Don't stop the BiBOP: Flexible and efficient storage for dynamically-typed languages. Technical report, Indiana University Computer Science Department, 1994.

Информация об авторах / Information about authors

Дмитрий Сергеевич КОСАРЕВ является ассистентом кафедры Системного программирования мат-мех. факультета СПбГУ. Его научные интересы включают функциональное программирование, компиляторы и реляционное программирование.

Dmitry Sergeevich KOSAREV is an assistant at the Chair of System Programming of Mathematics and Mechanics Faculty of SPBU. His research interests include functional programming, compilers, and relational programming.

Дмитрий Юрьевич БУЛЫЧЕВ – доцент кафедры системного программирования мат.-мех. факультета СПбГУ, кандидат физико-математических наук. Область научных интересов - языки и инструменты программирования, компиляторы, функциональное, логическое и реляционное программирование, анализ и синтез программ.

Dmitry Yuryevich BOULYTCHEV is an associate professor of the System Programming Chair of the Faculty of Mathematics and Mechanics of SPBU, Candidate of Physical and Mathematical Sciences. His research interests include programming languages and tools, compilers, functional, logical and relational programming, program analysis and synthesis.