

typedKanren: типизированное реляционное программирование в Haskell

Стариков Артем Игоревич

Научный руководитель: Кудасов Николай Дмитриевич

Университет Иннополис

JointMETA, 1 июля 2024 г.

miniKanren

Примеры программ на Scheme

Задачи перед статически типизированным диалектом

typedKanren

Класс типов **Logical**

Логическое представление типов

Унификация

Монада **Goal**

Пример: конкатенация списков

Реляционные версии сопоставления с образцом

Сравнение с другими реализациями

Заключение

miniKanren — это семейство логических языков программирования, встроенных в другие языки. Изначальную реализацию на языке Scheme представил Уильям Бирд в своей диссертации в 2009 году¹. Впоследствии появились реализации miniKanren на Clojure, Kotlin, OCaml, Python, Rust и других языках².

¹Byrd, “Relational Programming in miniKanren: Techniques, Applications, and Implementations”.

²<https://minikanren.org>

Унификация

Унификация переменной со значением:

```
> (run* (x)  
      (== x 1))  
'(1)
```

Унификация

Унификация переменной со значением:

```
> (run* (x)  
      (== x 1))  
'(1)
```

Свободная переменная в решении:

```
> (run* (x))  
'(_ . 0)
```

Унификация

Унификация переменной со значением:

```
> (run* (x)
    (== x 1))
' (1)
```

Свободная переменная в решении:

```
> (run* (x))
' ( _ .0)
```

Список со свободной переменной внутри:

```
> (run* (x)
    (fresh (y)
      (== x `(1 ,y)))))
' ((1 _ .0))
```

Конъюнкция и дизъюнкция

Противоречие в программе через конъюнкцию:

```
> (run* (x)
      (== x 1)
      (== x 2))
'()
```

Конъюнкция и дизъюнкция

Противоречие в программе через конъюнкцию:

```
> (run* (x)
      (== x 1)
      (== x 2))
'()
```

Несколько решений через дизъюнкцию:

```
> (run* (x)
      (conde
        [(== x 1)]
        [(== x 2)]))
'(1 2)
```


Пример: конкатенация списков

```
(defrel (appendo xs ys zs)
  (conde
    [(== xs '()) (== ys zs)]
    [(fresh (x xs^ zs^)
      (== xs `(,x . ,xs^))
      (== zs `(,x . ,zs^))
      (appendo xs^ ys zs^))]))
```

Пример: конкатенация списков

Запуск программы:

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

Пример: конкатенация списков

Запуск программы:

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

Запуск программы «в обратном направлении»:

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

Пример: конкатенация списков

Запуск программы:

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

Запуск программы «в обратном направлении»:

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

Нахождение всех способов разделить список на два:

```
> (run* (q r) (appendo q r '(1 2 3 4)))  
'(((()) (1 2 3 4))  
  ((1) (2 3 4))  
  ((1 2) (3 4))  
  ((1 2 3) (4))  
  ((1 2 3 4) ()))
```

Задачи перед typedKanren

Мы реализуем typedKanren — диалект miniKanren на Haskell: функциональном языке программирования со статической типизацией и ленивой моделью вычисления. При этом мы хотим достичь:

- ▶ поддержки основных операций: унификация, конъюнкция, дизъюнкция, создание свежих переменных и запуск реляционных программ;

Задачи перед typedKanren

Мы реализуем typedKanren — диалект miniKanren на Haskell: функциональном языке программирования со статической типизацией и ленивой моделью вычисления. При этом мы хотим достичь:

- ▶ поддержки основных операций: унификация, конъюнкция, дизъюнкция, создание свежих переменных и запуск реляционных программ;
- ▶ поддержки пользовательских типов в реляционных программах;

Задачи перед typedKanren

Мы реализуем typedKanren — диалект miniKanren на Haskell: функциональном языке программирования со статической типизацией и ленивой моделью вычисления. При этом мы хотим достичь:

- ▶ поддержки основных операций: унификация, конъюнкция, дизъюнкция, создание свежих переменных и запуск реляционных программ;
- ▶ поддержки пользовательских типов в реляционных программах;
- ▶ удобной записи реляционных программ с использованием обычных синтаксических конструкций Haskell;

Задачи перед typedKanren

Мы реализуем typedKanren — диалект miniKanren на Haskell: функциональном языке программирования со статической типизацией и ленивой моделью вычисления. При этом мы хотим достичь:

- ▶ поддержки основных операций: унификация, конъюнкция, дизъюнкция, создание свежих переменных и запуск реляционных программ;
- ▶ поддержки пользовательских типов в реляционных программах;
- ▶ удобной записи реляционных программ с использованием обычных синтаксических конструкций Haskell;
- ▶ высокой производительности реляционных программ, использующих typedKanren.

Пользовательские типы: класс типов `Logical`

Прежде всего объявим класс типов, которые можно использовать в реляционных программах:

```
class Logical a where
  type Logic a = r | r -> a
  unify :: Logic a -> Logic a -> State -> Maybe State
  walk  :: State -> Logic a -> Logic a
  inject :: a -> Logic a
  extract :: Logic a -> Maybe a
  ...
```

Пользовательские типы: класс типов `Logical`

Прежде всего объявим класс типов, которые можно использовать в реляционных программах:

```
class Logical a where
  type Logic a = r | r -> a
  unify :: Logic a -> Logic a -> State -> Maybe State
  walk  :: State -> Logic a -> Logic a
  inject :: a -> Logic a
  extract :: Logic a -> Maybe a
  ...
```

Также объявим `Term` а — тип, который содержит либо переменную, либо значение типа `Logic a`:

```
data Term a
  = Var (VarId a)
  | Value (Logic a)
```

Логическое представление типов

Простые типы, такие как числа и символы, не содержат в себе полей, а потому могут использоваться в реляционных программах как есть.

```
instance Logical Int where  
  type Logic Int = Int  
  ...
```

Логическое представление типов

Простые типы, такие как числа и символы, не содержат в себе полей, а потому могут использоваться в реляционных программах как есть.

```
instance Logical Int where
  type Logic Int = Int
  ...
```

Более сложные типы, как списки и деревья, содержат данные, и в реляционных программах бывает нужно заменить их переменной. Поэтому для таких типов нужно отдельное, *логическое* представление, которое разрешает использование переменных.

<pre>data Tree a = Empty Leaf a Node (Tree a) (Tree a)</pre>	<pre>data LTree a = LEmpty LLeaf (Term a) LNode (Term (Tree a)) (Term (Tree a))</pre>
--	---

Унификация

Метод `unify` класса `Logical` принимает два логических значения и пробует их унифицировать.

```
unify :: Logical a
      => LTree a -> LTree a -> State -> Maybe State
unify LEmpty      LEmpty      state = Just state
unify (LLeaf x)   (LLeaf y)   state = unify' x y state
unify (LNode xl xr) (LNode yl yr) state = do
  state' <- unify' xl yl state
  unify' xr yr state'
unify _ _ _ = Nothing
```

Унификация

Пользовательский `unify`, тем не менее, не работает с переменными напрямую. Унификация логических переменных делегируется функции `unify'`.

```
unify' :: Logical a
      => Term a -> Term a -> State -> Maybe State
unify' l r state =
  case (walk' state l, walk' state r) of
    (Var x, Var y) | x == y -> Just state
    (Var x, r') -> addSubst x r' state
    (l', Var y) -> addSubst y l' state
    (Value l', Value r') -> unify l' r' state
```

Монада Goal

При написании реляционных программ понадобится монада `Goal`. Она оборачивает функцию, принимающее состояние и возвращающее поток состояний.

```
newtype Goal x = Goal (State -> Stream (State, x))
```

Так как `Goal` является монадой, мы можем использовать `do`-нотацию при написании реляционных программ, что значительно улучшает их читаемость.

Основные операции над Goal

`(==)` :: Logical a => Term a -> Term a -> Goal () —

унифицирует два логических значения;

`conj` :: Goal x -> Goal y -> Goal y — конъюнкция двух целей;

`disj` :: Goal x -> Goal x -> Goal x — дизъюнкция двух целей;

`conjMany` :: [Goal ()] -> Goal () — конъюнкция списка целей;

`disjMany` :: [Goal x] -> Goal x — дизъюнкция списка целей;

`fresh` :: Fresh v => Goal v создает логические переменные
в нужном пользователю количестве;

`run` :: Fresh v => (v -> Goal ()) -> [v] запускает реляционную
программу и возвращает список решений.

Пример на typedKanren: конкатенация списков

```
appendo :: Logical a
          => Term [a] -> Term [a] -> Term [a] -> Goal ()
appendo xs ys zs = disjMany
  [ do
    xs === Value LogicNil
    ys === zs
  , do
    (x, xs', zs') <- fresh
    xs === Value (LogicCons x xs')
    zs === Value (LogicCons x zs')
    appendo xs' ys zs'
  ]
```

`LogicNil` и `LogicCons` являются логическими аналогами конструкторов `[]` и `(:)` соответственно.

Пример на typedKanren: конкатенация списков

Запуск программы:

```
>>> run (\q -> appendo [1, 2] [3, 4 :: Term Int] q)  
[[1,2,3,4]]
```

Пример на typedKanren: конкатенация списков

Запуск программы:

```
>>> run (\q -> appendo [1, 2] [3, 4 :: Term Int] q)  
[[1,2,3,4]]
```

Запуск программы «в обратном направлении»:

```
>>> run (\q -> appendo [1, 2] q [1, 2, 3, 4 :: Term Int])  
[[3,4]]
```

Пример на typedKanren: конкатенация списков

Запуск программы:

```
>>> run (\q -> appendo [1, 2] [3, 4 :: Term Int] q)
[[1,2,3,4]]
```

Запуск программы «в обратном направлении»:

```
>>> run (\q -> appendo [1, 2] q [1, 2, 3, 4 :: Term Int])
[[3,4]]
```

Нахождение всех способов разделить список на два:

```
>>> mapM_ print $
...   run (\(q, r) -> appendo q r [1, 2, 3, 4 :: Term Int])
([ ], [1,2,3,4])
([1], [2,3,4])
([1,2], [3,4])
([1,2,3], [4])
([1,2,3,4], [ ])
```

Реляционные версии сопоставления с образцом

Заметим, что `conde` в `miniKanren` — это реляционная версия конструкции `cond`:

```
(defrel (appendo xs ys zs)
  (conde
    [(== xs '()) (== ys zs)]
    [(fresh (x xs^ zs^)
      (== xs `(,x . ,xs^))
      (== zs `(,x . ,zs^))
      (appendo xs^ ys zs^))]))
```

```
(define (append xs ys)
  (cond
    [(eq? xs '()) ys]
    [else (let
      ((x (car xs)) (xs^ (cdr xs)))
      (cons x
        (append xs^ ys)))]))
```

Реляционные версии сопоставления с образцом

В реализации `append` на Haskell вместо ветвлений идиоматично использовать *сопоставление с образцом*:

```
append :: [a] -> [a] -> [a]
append xs ys = case xs of
  [] -> ys
  (x:xs') -> x : append xs' ys
```

Поэтому при реализации `appendo` на Haskell также хотелось бы использовать сопоставление с образцом вместо `disjMany`.

Реляционные версии сопоставления с образцом

В реализации `append` на Haskell вместо ветвлений идиоматично использовать *сопоставление с образцом*:

```
append :: [a] -> [a] -> [a]
append xs ys = case xs of
    [] -> ys
    (x:xs') -> x : append xs' ys
```

Поэтому при реализации `appendo` на Haskell также хотелось бы использовать сопоставление с образцом вместо `disjMany`.

К сожалению, использовать встроенное выражение `case` в реляционных программах не получится. `case` выбирает первый подходящий образец, тогда как в реляционных программах необходимо рассмотреть каждый образец. Haskell не позволяет изменить это поведение.

Реляционные версии сопоставления с образцом

При реализации реляционной версии `case` мы ориентировались на библиотеку `total`³, которая позволяет проводить исчерпывающее сопоставление с образцом с помощью *призм*⁴:

```
total :: Either Char Int -> String
total = _case
  & on _Left (\c -> replicate 3 c)
  & on _Right (\n -> replicate n '!')
```

```
total = \case
  Left c -> replicate 3 c
  Right n -> replicate n '!'
```

³Gonzalez, *total-1.0.0: Exhaustive pattern matching using traversals, prisms, and lenses*; Gonzalez, *total library*.

⁴Laarhoven, *CPS based functional references*; Kmett, *lens library*.

Реляционные версии сопоставления с образцом: неисчерпывающий вариант

В `typedKanren` реализованы две версии сопоставления с образцом: неисчерпывающая и исчерпывающая. Неисчерпывающая является более простой и позволяет использовать призмы, сгенерированные с помощью уже существующей библиотеки `lens`⁵.

```
appendo xs ys zs = xs & (matche
  & on _LogicNil (\() -> ys === zs)
  & on _LogicCons (\(x, xs') -> do
    zs' <- fresh
    zs === LogicCons x zs'
    appendo xs' ys zs'))
```

⁵Kmett, *lens library*.

Реляционные версии сопоставления с образцом: исчерпывающий вариант

Исчерпывающая версия сопоставления позволяет на этапе компиляции убедиться, что проверены все варианты. Для этого необходимы специальные версии призм, в типе которых содержится информация, какой вариант из всех доступных проверяет эта призма. В остальном исчерпывающая версия не отличается от неисчерпывающей.

```
appendo xs ys zs = xs & (matche'  
  & on' _LogicNil' (\() -> ys === zs)  
  & on' _LogicCons' (\(x, xs') -> do  
    zs' <- fresh  
    zs === LogicCons x zs'  
    appendo xs' ys zs')  
  & enter')
```

Сравнение с другими реализациями

Мы сравнили производительность `typedKanren` с реализациями `miniKanren` на других языках: `faster-miniKanren`⁶ на Racket, `OCanren`⁷ на OCaml и `klogic`⁸ на Kotlin. Последние две отличаются тем, что встроены в статически типизированный язык и поддерживают пользовательские типы.

⁶Ballantyne, *A fast implementation of miniKanren with disequality and absento, compatible with Racket and Chez*.

⁷Kosarev and Boulytchev, "Typed embedding of a relational language in OCaml".

⁸Kamenev et al., "klogic: miniKanren in Kotlin".

Сравнение с другими реализациями

Для сравнения мы использовали следующие программы:

- ▶ Вычисление 3^5 и $\log_3 243$ в двоичной системе счисления⁹;
- ▶ Нахождение 100 квайнов первого порядка (quines), 15 квайнов второго порядка (twines) и 2 квайна третьего порядка (thrines) для реляционного интерпретатора Scheme¹⁰.

Исходный код всех программ доступен в репозитории на GitHub¹¹.

⁹Kiselyov et al., “Pure, Declarative, and Constructive Arithmetic Relations (Declarative Pearl)”.

¹⁰Byrd, Holk, and Friedman, “miniKanren, Live and Untagged: Quine Generation via Relational Interpreters (Programming Pearl)”.

¹¹<https://github.com/SnejUgal/typedKanren-benchmarks>

Сравнение с другими реализациями

	3^5	$\log_3 243$	quines	twines	thrines
faster-miniKanren	118.8	20.3	332.0	285.4	529.5
OCanren	463.1	65.9	832.3	759.8	1 317.9
klogic	1 189.8	91.0	1 015.0	1 227.5	5 346.5
typedKanren	690.8	83.5	1 378.0	1 907.0	4 462.0

Figure: Результаты сравнения производительности. Указано время в миллисекундах.

typedKanren сравним по производительности с klogic, но медленнее остальных реализаций. Мы планируем улучшить производительность typedKanren, применив оптимизации из других реализаций miniKanren.

Заключение

Мы реализовали `typedKanren` — диалект `miniKanren` на Haskell. Как и другие диалекты, `typedKanren` поддерживает основные операции `miniKanren` и реализует честный поиск. От большинства других диалектов `typedKanren` отличает поддержка статической типизации, а также наличие реляционного сопоставления с образцом.

Заключение

Мы реализовали `typedKanren` — диалект `miniKanren` на Haskell. Как и другие диалекты, `typedKanren` поддерживает основные операции `miniKanren` и реализует честный поиск. От большинства других диалектов `typedKanren` отличает поддержка статической типизации, а также наличие реляционного сопоставления с образцом.

Мы сравнили производительность `typedKanren` с некоторыми диалектами `miniKanren` на других языках. Наша реализация оказалась сравнимой с `klogic`, но медленнее других реализаций.

Мы рассматриваем возможности оптимизации с применением более эффективных алгоритмов.




Заключение

В дальнейшем мы планируем реализовать квазиквотирование для упрощения синтаксиса реляционных программ, а также исследовать автоматическую конвертацию функциональных программ в реляционные.





Исходный код `typedKanren` доступен на GitHub¹².

¹²<https://github.com/SnejUgal/typedKanren>




Список литературы I

-  Ballantyne, Michael. *A fast implementation of miniKanren with disequality and absento, compatible with Racket and Chez*. 2015. URL: <https://github.com/michaelballantyne/faster-minikanren> (visited on 06/06/2024).
-  Byrd, William E. “Relational Programming in miniKanren: Techniques, Applications, and Implementations”. AAI3380156. PhD thesis. USA, 2009. ISBN: 9781109504682.
-  Byrd, William E., Eric Holk, and Daniel P. Friedman. “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. Copenhagen, Denmark: Association for Computing Machinery, 2012, pp. 8–29. ISBN: 9781450318952. DOI: 10.1145/2661103.2661105. URL: <https://doi.org/10.1145/2661103.2661105>.

Список литературы II

-  Gonzalez, Gabriella. *total library*. URL: <https://hackage.haskell.org/package/total> (visited on 06/06/2024).
-  — *.total-1.0.0: Exhaustive pattern matching using traversals, prisms, and lenses*. Jan. 2015. URL: <https://www.haskellforall.com/2015/01/total-100-exhaustive-pattern-matching.html> (visited on 06/06/2024).
-  Kamenev, Yury et al. “klogic: miniKanren in Kotlin”. In: *Proceedings of the Fifth miniKanren Workshop (miniKanren '23)*. 2023. URL: <http://minikanren.org/workshop/2023/minikanren23-final4.pdf>.
-  Kiselyov, Oleg et al. “Pure, Declarative, and Constructive Arithmetic Relations (Declarative Pearl)”. In: *Functional and Logic Programming*. Ed. by Jacques Garrigue and Manuel V. Hermenegildo. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 64–80. ISBN: 978-3-540-78969-7.

Список литературы III

-  Kmett, Edward. *lens library*. URL: <https://hackage.haskell.org/package/lens> (visited on 06/06/2024).
-  Kosarev, Dmitrii and Dmitry Boulytchev. “Typed embedding of a relational language in OCaml”. In: *arXiv preprint arXiv:1805.11006* (2018).
-  Laarhoven, Twan van. *CPS based functional references*. July 2009. URL: <https://www.twanvl.nl/blog/haskell/cps-functional-references> (visited on 06/06/2024).