

Реляционный синтез сопоставления с образцом

Relational Synthesis for Pattern Matching

Косарев Дмитрий

Будет опубликовано на
Asian Symposium on Programming Languages and Systems (APLAS) 2020

9 ноября 2020

Build date: November 9, 2020

Существенная часть функционального программирования

Два основных подхода:

- диаграммы решений
 - минимизируется количество проверок
- автомат с возвратами
 - минимизируется размер кода
- **синтез программ** (наш подход)

1 Обзор

2 Заслуги работы

- Синтез с помощью реляционных интерпретаторов
- Создание конечного набора примеров
- Оптимизации

3 Реализация

4 Проблемы

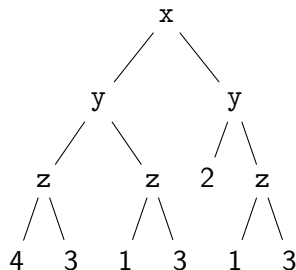
5 Заключение

Пример: использование диаграмм решений

match x,y,z with

```
| _,F,T → 1  
| F,T,_ → 2  
| _,_ ,F → 3  
| _,_ ,T → 4
```

```
if x then  
  if y then  
    if z then 4 else 3  
  else  
    if z then 1 else 3  
else  
  if y then 2  
  else  
    if z then 1 else 3
```

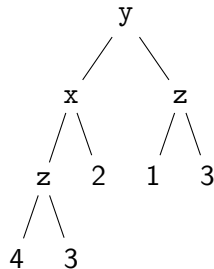


Пример: использование диаграмм решений

match x,y,z with

```
| _,F,T → 1  
| F,T,_ → 2  
| _,_ ,F → 3  
| _,_ ,T → 4
```

```
if y then  
  if x then  
    if z then 4 else 3  
  else 2  
else  
  if z then 1 else 3
```



Пример: Компиляция в автомат с возвратами

$$(P \rightarrow L) = \left(\begin{array}{cccl} [] & - & \rightarrow & 1 \\ - & [] & \rightarrow & 2 \\ x::xs & y::ys & \rightarrow & 3 \end{array} \right)$$

Пример: Компиляция в автомат с возвратами

$$(P \rightarrow L) = \begin{pmatrix} [] & _ & \rightarrow & 1 \\ _ & [] & \rightarrow & 2 \\ x::xs & y::ys & \rightarrow & 3 \end{pmatrix}$$

$$(P_1 \rightarrow L_1) = ([[] \quad _ \rightarrow 1)$$

$$(P_2 \rightarrow L_2) = (_ \quad [] \rightarrow 2)$$

$$(P_3 \rightarrow L_3) = (x::xs \quad y::ys \rightarrow 3)$$

Пример: Компиляция в автомат с возвратами

$$(P \rightarrow L) = \begin{pmatrix} [] & - & \rightarrow 1 \\ - & [] & \rightarrow 2 \\ x::xs & y::ys & \rightarrow 3 \end{pmatrix}$$

$$(P_1 \rightarrow L_1) = ([[] \quad - \rightarrow 1)$$

$$(P_2 \rightarrow L_2) = (- \quad [[] \rightarrow 2)$$

$$(P_3 \rightarrow L_3) = (x::xs \quad y::ys \rightarrow 3)$$

```
(switch lx with case []: 1
  default:      )
```

```
(switch ly with case []: 2
  default:      )
```

```
(switch lx with
  case (::):
    (switch ly with
      case (::) : 3
      default:  )
  default:      )
```


Пример: Компиляция в автомат с возвратами

$$(P \rightarrow L) = \begin{pmatrix} [] & - & \rightarrow 1 \\ - & [] & \rightarrow 2 \\ x::xs & y::ys & \rightarrow 3 \end{pmatrix}$$

$$(P_1 \rightarrow L_1) = ([[] \quad - \rightarrow 1)$$

$$(P_2 \rightarrow L_2) = (- \quad [[] \rightarrow 2)$$

$$(P_3 \rightarrow L_3) = (x::xs \quad y::ys \rightarrow 3)$$

```
catch
  (catch
    (switch lx with case []: 1
      default: exit)
    with (catch
      (switch ly with case []: 2
        default: exit)
      with (catch
        (switch lx with
          case (::):
            (switch ly with
              case (::) : 3
              default: exit)
          default: exit))))
    with (failwith "Partial match")
```

Пример: Компиляция в автомат с возвратами + оптимизация

$$(P \rightarrow L) = \left(\begin{array}{lll} [] & \bar{-} & \rightarrow 1 \\ - & [] & \rightarrow 2 \\ x::xs & y::ys & \rightarrow 3 \end{array} \right)$$

Пример: Компиляция в автомат с возвратами + оптимизация

$$(P \rightarrow L) = \begin{pmatrix} [] & - & \rightarrow & 1 \\ x::xs & y::ys & \rightarrow & 3 \\ - & [] & \rightarrow & 2 \end{pmatrix}$$

Пример: Компиляция в автомат с возвратами + оптимизация

$$(P \rightarrow L) = \begin{pmatrix} [] & - & \rightarrow & 1 \\ \mathbf{x}::\mathbf{x}\mathbf{s} & \mathbf{y}::\mathbf{y}\mathbf{s} & \rightarrow & 3 \\ - & [] & \rightarrow & 2 \end{pmatrix}$$

$$(P_1 \rightarrow L_1) = \begin{pmatrix} [] & - & \rightarrow & 1 \\ \mathbf{x}::\mathbf{x}\mathbf{s} & \mathbf{y}::\mathbf{y}\mathbf{s} & \rightarrow & 3 \end{pmatrix}$$

$$(P_2 \rightarrow L_2) = (- \quad [] \rightarrow 2)$$

Пример: Компиляция в автомат с возвратами + оптимизация

$$(P \rightarrow L) = \begin{pmatrix} [] & - & \rightarrow 1 \\ x::xs & y::ys & \rightarrow 3 \\ - & [] & \rightarrow 2 \end{pmatrix}$$

$$(P_1 \rightarrow L_1) = \begin{pmatrix} [] & - & \rightarrow 1 \\ x::xs & y::ys & \rightarrow 3 \end{pmatrix}$$

$$(P_2 \rightarrow L_2) = (- \quad [] \rightarrow 2)$$

```
(switch lx with
  case []: 1
  case (::) :
    (switch ly with
      case (::): 3
      default:  ))
```

```
(switch ly with
  case []: 2
  default:  )
```

Пример: Компиляция в автомат с возвратами + оптимизация

$$(P \rightarrow L) = \begin{pmatrix} [] & - & \rightarrow 1 \\ x::xs & y::ys & \rightarrow 3 \\ - & [] & \rightarrow 2 \end{pmatrix}$$

$$(P_1 \rightarrow L_1) = \begin{pmatrix} [] & - & \rightarrow 1 \\ x::xs & y::ys & \rightarrow 3 \end{pmatrix}$$

$$(P_2 \rightarrow L_2) = (- \quad [] \rightarrow 2)$$

```
catch
  (catch
    (switch lx with
      case []: 1
      case (::) :
        (switch ly with
          case (::): 3
          default: exit))
    with
      (switch ly with
        case []: 2
        default: exit)
    with (failwith "Partial match"))
```

Ещё оптимизации для автоматов с возвратами

- Использование информации о полноте
- Оптимизация конструкций `exit`, новый синтаксис `exit N`
 - Дешёвая поддержка охранных выражений (pattern guards)
- Ещё кое-что, для избегания повторных проверок

В компиляторе OCaml используется реализация из [LFM01].

```
case x,y of
  True, False → 1
  True, True  → 2
  False,False → 3
  False,True  → 4
```


Языки с ленивой семантикой

```
f :: Bool → Bool  
f True = True  
f x     = f x
```

```
y :: Bool  
y = f x
```

```
case x,y of  
  True, False → 1  
  True, True  → 2  
  False,False → 3  
  False,True  → 4
```

Языки с ленивой семантикой

```
f :: Bool → Bool  
f True = True  
f x     = f x
```

```
y :: Bool  
y = f x
```

```
case x,y of  
  True, False → 1  
  True, True  → 2  
  False,False → 3  
  False,True  → 4
```

В *ленивых* языках у нас нет свободы
в выборе первого столбца \Rightarrow в *строгих*
больше простора для оптимизаций

Понятие "хорошей" скомпилированной программы

В литературе упоминаются [SR00] следующие эвристики

- Слева-направо
- Справа-налево
- Small branching factor (малый коэффициент ветвления)
- Large branching factor
- Leaf edges
- Arity factor
- Artificial rule
- и другие

В подавляющем большинстве случаев формально более оптимальные программы показывают сходную производительность

1 Обзор

2 Заслуги работы

- Синтез с помощью реляционных интерпретаторов
- Создание конечного набора примеров
- Оптимизации

3 Реализация

4 Проблемы

5 Заключение

- I Спроектировали синтез с помощью комбинации *реляционных интерпретаторов* на miniKanren
- II Заменяли \forall на *конечный* набор примеров
- III Сделали оптимизацию методом ветвей и границ с помощью нового примитива miniKanren: *ограничение на структуру (structural constraint)*

1 Обзор

2 Заслуги работы

- Синтез с помощью реляционных интерпретаторов
- Создание конечного набора примеров
- Оптимизации

3 Реализация

4 Проблемы

5 Заключение

Реляционные интерпретаторы

Семейство языков miniKanren – реинкарнация логического программирования
Программы представляются как вычисляемые отношения ("реляции", англ. relation)

$$\text{Interpret} : \text{Program} \times \text{Input} \times \text{Result}$$

Их можно запускать¹ в разные стороны:

$$\text{Interpret}_{\rightarrow} : \text{Program} \times \text{Input} \rightarrow \text{Result}$$

$$\text{Synthesize}_{\leftarrow} : \text{Input} \times \text{Result} \rightarrow \text{Program}$$

¹Если интерпретатор написан достаточно аккуратно

Синтаксис двух языков сопоставления с образцом

Язык сопоставления с образцом

$$\mathcal{C} = \{C_1^{k_1}, \dots, C_n^{k_n}\}$$

$$\mathcal{V} = \mathcal{C}\mathcal{V}^*$$

$$\mathcal{P} = _ | \mathcal{C}\mathcal{P}^*$$

$$\langle v; p_1, \dots, p_k \rangle \longrightarrow i \\ 1 \leq i \leq k + 1$$

Язык скомпилированного представления

$$\mathcal{M} = \bullet$$

$$\mathcal{M}[\mathbb{N}]$$

$$\mathcal{S} = \textbf{return } \mathbb{N}$$

$$\textbf{switch } \mathcal{M} \textbf{ with } [\mathcal{C} \rightarrow \mathcal{S}]^* \textbf{ otherwise } \mathcal{S}$$

Опущены для простоты: типы, охранные выражения, переменные в паттернах и т.д.

Синтаксис двух языков сопоставления с образцом

$$match^o : \mathcal{V} \times \mathcal{P}^* \times \mathbb{N}$$

Язык сопоставления с образцом

$$\mathcal{C} = \{C_1^{k_1}, \dots, C_n^{k_n}\}$$

$$\mathcal{V} = \mathcal{C} \mathcal{V}^*$$

$$\mathcal{P} = _ \mid \mathcal{C} \mathcal{P}^*$$

$$\langle v; p_1, \dots, p_k \rangle \longrightarrow i \\ 1 \leq i \leq k + 1$$

$$eval_S^o : \mathcal{V} \times \mathcal{S} \times \mathbb{N}$$

Язык скомпилированного представления

$$\mathcal{M} = \bullet$$

$$\mathcal{M} [\mathbb{N}]$$

$$\mathcal{S} = \textbf{return } \mathbb{N}$$

$$\textbf{switch } \mathcal{M} \textbf{ with } [\mathcal{C} \rightarrow \mathcal{S}]^* \textbf{ otherwise } \mathcal{S}$$

Опущены для простоты: типы, охранные выражения, переменные в паттернах и т.д.

$$\forall v \quad \forall (1 \leq i \leq k+1) \quad (match^o v p_1, \dots, p_k i) \Leftrightarrow eval_S^o v \textcircled{?} i$$

- $match^o v p_1, \dots, p_k i$ – интерпретатор языка сопоставления с образцом, для каждого сопоставляемого выражения (scrutinee) v выдает номер ветви i
- $\textcircled{?}$ – программа, которую надо синтезировать
- $eval_S^o v \textcircled{?} i$ – а интерпретатор скомпилированного представления S , которая проверяет, что синтезированная программа $\textcircled{?}$ на примерах v выдает правильные номера ветвей i

👍 Интерпретаторы $match^o$ и $eval^o$ легко реализовать (и для расширений сопоставления с образцом тоже должно быть легко)

👎 miniKanren с disequality constraints не умеет работать с кванторами \forall

1 Обзор

2 Заслуги работы

- Синтез с помощью реляционных интерпретаторов
- Создание конечного набора примеров
- Оптимизации

3 Реализация

4 Проблемы

5 Заключение

Избавление от \forall . Создание конечного набора примеров

Для каждого сопоставления с образцом мы знаем:

- тип сопоставляемого выражения
- все образцы, которые используются
 - можем посчитать максимальную глубину образцов

Идея

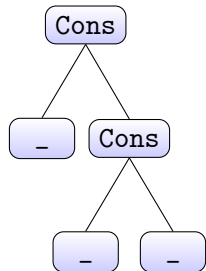
Переберём всех жителей типа сопоставляемого выражения до некоторой глубины, и будем использовать этих жителей как примеры



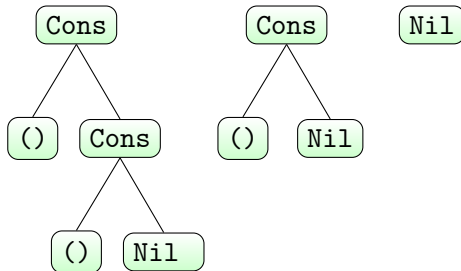
В худшем случае — экспоненциальное количество примеров

Пример 1: Полный набор примеров из трёх штук

```
match (s : unit list) with  
| _ :: _ :: _ → 1  
| _           → 2
```



(a) Два образца



(b) Три примера

Пример 2: Недостаточно полный набор примеров

Два образца:

Nil

-

Набор примеров, ограниченных глубиной 1, состоит только из одного примера

Nil

```
match (s : unit list) with
```

```
| [] → 1
```

```
| - → 2
```

Данная программа ведет себя согласовано на наборе примеров

switch ... with

| Nil → 1

| **otherwise** → 1

но, очевидно, неправильная

Текущий алгоритм для получения примеров

Если кратко:

- Вычислить глубину образцов h
- Синтезируем всех жителей, но
- на глубине $h + 1$ используем заранее подготовленного жителя соответствующего типа



В худшем случае — экспоненциальное количество примеров

1 Обзор

2 Заслуги работы

- Синтез с помощью реляционных интерпретаторов
- Создание конечного набора примеров
- **Оптимизации**

3 Реализация

4 Проблемы

5 Заключение

Оптимизация: откидывание эквивалентных программ

Очевидно, что реляционный интерпретатор языка \mathcal{S} может перебирать различные эквивалентные программы

```
switch • with  
true  → ...  
false → ...
```



```
switch • with  
false → ...  
true  → ...
```

С этой проблемой удалось побороться

- Задав порядок на ветвях **switch**ей, используя информацию о типе
- Это несколько «сломало» интерпретатор, но для синтеза это не существенно

Оптимизация: сокращение необходимого набора примеров

Пример:

```
match (s : bool * bool * bool) with  
| (_,_,F) → 1  
| (_,_,T) → 2
```

Мы можем во время компиляции обнаружить, что

- Всего 2^3 жителей у типа `bool * bool * bool`
- Не надо проверять, что сопоставляемое выражение — это тройка
- Не надо заглядывать в 1ю и 2ю компоненты, так как важна только третья

Итого, может запускать синтез только на двух примерах $\{(B, B, T), (B, B, F)\}$ (где B — это любое значение типа `bool`) если мы **запретим "заглядывание"** в ненужные **поддеревья** примеров

Оптимизация методом ветвей и границ

Текущий результат синтеза хранится в (?) и во время поиска мы *только уточняем* результат, добавляя новые конструкции **switch**

Идея

Если текущее приближение ответа длиннее, чем уже найденный ответ — прерываем поиск в этой ветке

Требует модификации примитива `miniKanren run`: для каждого найденного ответа

- считаем размер
- обновляем минимальный найденный ответ

Отсечение ветвей поиска делается помощью нового примитива — *ограничения на структуру* (*structural constraint*)

Ограничение на структуру (structural constraint)

Новый примитив:

- Принимает промежуточное представление значений и конвертирует их (в текущем состоянии) до логических
- Принимает предикат для логических значений
- Если результат слишком большой — прекратить поиск(*failure*^o)
- Иначе продолжать поиск, не меняя состояние (*success*^o)

Особенности:

- Используется для вычисления размера текущего решения
- Может учитывать или не учитывать disequality constraints
- Можно использовать, чтобы реализовать *absent*^o и подобные ограничения
- Работает с логическим представлением (reified) значений \Rightarrow медленно

Критерий минимизации для синтезированных программ

switch \mathcal{M} **with**

$\mathcal{C}_1 \rightarrow \mathcal{S}_1$

...

$\mathcal{C}_n \rightarrow \mathcal{S}_n$

otherwise \mathcal{S}



if $\mathcal{M} = \mathcal{C}_1$ **then** \mathcal{S}_1

...

else if $\mathcal{M} = \mathcal{C}_n$ **then** \mathcal{S}_n

else \mathcal{S}

Интуиция: один **switch** с n случаев можно примерно закодировать в n **if**ов

Будем считать, что размер

- **switch** — это число веток
- **return** равен 0
- программы целиком — сумма размеров всех входящих в неё **switch**

Наш критерий минимизации: уменьшение размера синтезированной программы

Но могут быть другие: глубина, коэффициент ветвления, и т.д.

1 Обзор

2 Заслуги работы

- Синтез с помощью реляционных интерпретаторов
- Создание конечного набора примеров
- Оптимизации

3 Реализация

4 Проблемы

5 Заключение

Мы пользуемся OCanren — типобезопасным встраиванием miniKanren в OCaml.

В процессе используется noCanren [LVB17] для порождения кода на OCanren [KB16, OCa]

Основная часть (два реляционных интерпретатора + порождение примеров) сделаны с помощью noCanren.

Репозиторий проекта [Rep]

1 Обзор

2 Заслуги работы

- Синтез с помощью реляционных интерпретаторов
- Создание конечного набора примеров
- Оптимизации

3 Реализация

4 Проблемы

5 Заключение

Пример синтезированной программы: сопоставление выражения типа `bool*bool*bool`

```
match • with
  (_, false, true) → 1
  (false, true, _) → 2
  (_, _, false) → 3
  (_, _, true) → 4
```

Ответ размера 6 (за 1.6 секунд)

```
switch •[0] with
| true →
  (switch •[1] with
  | true →
    (switch •[2] with true → 4 | _ → 3)
  | _ →
    (switch •[2] with true → 1 | _ → 3))
| _ →
  (switch •[1] with
  | true → 2
  | _ →
    (switch •[2] with true → 1 | _ → 3))
```

Пример синтезированной программы: сопоставление выражения типа `bool*bool*bool`

`match • with`

`(_, false, true) → 1`

`(false, true, _) → 2`

`(_, _, false) → 3`

`(_, _, true) → 4`

Ответ размера 5 (за +0.4 секунд)

`switch • [0] with`

`| true →`

`(switch • [2] with`

`| true →`

`(switch • [1] with true → 4 | _ → 1)`

`| _ → 3)`

`| _ →`

`(switch • [1] with`

`| true → 2`

`| _ → (switch • [2] with true → 1 | _ → 3)`

Пример синтезированной программы: сопоставление выражения типа `bool*bool*bool`

```
match • with
  (_, false, true) → 1
  (false, true, _) → 2
  (_, _, false) → 3
  (_, _, true) → 4
```

Ответ размера 4 (за +0.7 секунд)

```
switch •[1] with
| true →
  (switch •[0] with
  | true →
    (switch •[2] with true → 4 | _ → 3)
  | _ → 2)
| _ →
  (switch •[2] with true → 1 | _ → 3)
```

Проблемы с производительностью (1/2)

```
match a,s,c with
| (_,_,Ldi i::_) → 1
| (_,_,Push::_) → 2
| (Int _,Val (Int _)::_,IOp _::_) → 3
| (Int _::_ ,Test (_,_)::c) → 4
| (Int _::_ ,Test (_,_)::c) → 5
| (_,_,Extend::_) → 6
| (_,_,Search _::_) → 7
| (_,_,Pushenv::_) → 8
| (_,Env e::s,Popenv::_) → 9
| (_,_,Mkclos cc::_) → 10
| (_,_,Mkclosrec _::_) → 11
| (Clo (_,_), Val _::_ , Apply::_) → 12
| (_, (Code _::Env _::_), []) → 13
| (_, [], []) → 14
```

Интерпретатор PCF (mini-ML)
из статьи Г.Плоткина, 1977

Сейчас не работает, потому
что слишком много (11102)
примеров

- большая глубина образцов
- много конструкторов в типах

Проблемы с производительностью (2/2)

```
type code =  
| Push  
| Ldi of int  
| IOp of int  
| Int of int  
type prog = code list  
type item =  
| Val of code  
| Env of int  
| Code of int  
type stack = item list
```

```
match (code, stack, prog) with  
| (_, _, (Ldi _)::_) → 1  
| (_, _, (Push _)::_) → 2
```

Сокращённый пример

- по типам
- по веткам

Для двух веток надо 5 примеров

Проблемы с производительностью (2/2)

```
type code =  
| Push  
| Ldi of int  
| IOp of int  
| Int of int  
type prog = code list  
type item =  
| Val of code  
| Env of int  
| Code of int  
type stack = item list  
  
match (code, stack, prog) with  
| (_, _, (Ldi _)::_) → 1  
| (_, _, (Push _)::_) → 2  
| (Int _, _, (IOp _)::_) → 3
```

Сокращённый пример

- по типам
- по веткам

Для двух веток надо 5 примеров

Для трёх веток и тех же типов уже необходимо 20 примеров

- за 1,5с получим 1й ответ размера 7
- ещё через полсекунды — 2й и 3й (последний) ответы размера 6 и 5, соответственно
- в конце оно тратит 10с, чтобы доказать, что более коротких ответов не существует

1 Обзор

2 Заслуги работы

- Синтез с помощью реляционных интерпретаторов
- Создание конечного набора примеров
- Оптимизации

3 Реализация

4 Проблемы

5 Заключение

Достижения:

- I Спроектировали синтез с помощью комбинации *реляционных интерпретаторов* на miniKanren
- II Заменяли \forall на *конечный* набор примеров
- III Сделали оптимизацию методом ветвей и границ с помощью нового примитива miniKanren: *ограничение на структуру (structural constraint)*







На маленьких примерах подход работает корректно, ... но на больших есть проблемы с производительностью

Пути дальнейшего улучшения

- Построение входного логического значения для ограничений на структуру можно делать эффективнее (ленивые вычисления)
- Использование конечнодоменных ограничений вместо disequality constraints
- Мемоизация сейчас никак не используется, т.к. disequality constraints
- Создание меньшего числа примеров
- Другое представление языка \mathcal{S} с использованием конструкций **exit**
 - тут может потребоваться номинальная унификация

Спасибо!

Литература

-  Dmitry Kosarev and Dmitry Boulytchev, *Typed embedding of a relational language in OCaml*, 1–22.
-  Fabrice Le Fessant and Luc Maranget, *Optimizing pattern matching*, SIGPLAN Not. **36** (2001), no. 10, 26–37.
-  Petr Lozov, Andrei Vyatkin, and Dmitry Boulytchev, *Typed relational conversion*, Trends in Functional Programming - 18th International Symposium, TFP 2017, Canterbury, UK, June 19-21, 2017, Revised Selected Papers, 2017, pp. 39–58.
-  *OCanren*, <https://github.com/JetBrains-Research/OCanren>, Accessed: 2020-05-18.
-  *Work in progress implementaiton*, <https://github.com/kakadu/pat-match>.
-  Kevin D Scott and Norman Ramsey, *When do match-compilation heuristics matter?*, 2000.