Реляционный синтез сопоставления с образцом

Relational Synthesis for Pattern Matching

Косарев Дмитрий

Опубликовано на IFCP 2020

9 ноября 2020

Build date: February 25, 2021

111

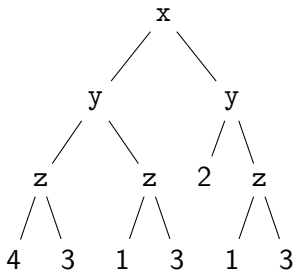
Template is Kakadu.

Оглавление

Пример: использование диаграмм решений

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

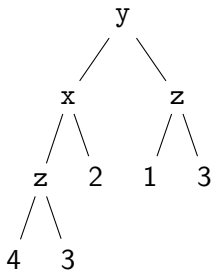
```
if x then
  if y then
    if z then 4 else 3
  else
    if z then 1 else 3
else
  if y then 2
  else
    if z then 1 else 3
```



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
```



Оглавление

Оглавление

Это кусок про optimization and analysis

```
(f <$> p) <*> (g <$> q)
```

$$\Downarrow$$

```
(\x y -> (f x) (g y)) <$> p <*> q
```

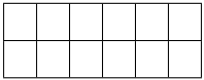
```
string ''ab''   ===   char 'a' <:> char 'b' <:> pure []
```

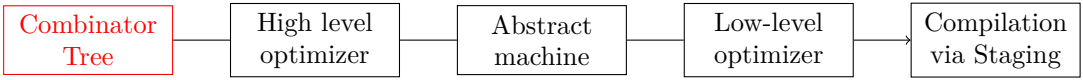
$$\Downarrow$$

```
satisfy (== 'a') *> satisfy (== 'b') $> ''ab''
```

Оглавление

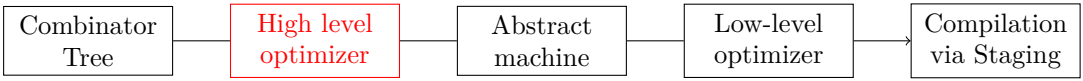
Parsec vs Parsley






```
newtype Fix (syn :: (* -> *) -> (* -> *)) (a :: *) where
  In :: syn (Fix syn) a -> Fix syn a

newtype Parser a = Parser (Fix ParserF a)
data ParserF (k :: * -> *) (a :: *) where
  Pure :: a -> ParserF k a
  Satisfy :: (Char -> Bool) -> ParserF k Char
  Try :: k a -> ParserF k a
  Look :: k a -> ParserF k a
  NegLook :: k () -> ParserF k ()
  (:<*>:) :: k (a -> b) -> k a -> ParserF k b
  (:*>:) :: k a -> k b -> ParserF k b
  (:<*:) :: k a -> k b -> ParserF k a
  (:<|>:) :: k a -> k a -> ParserF k a
  Empty :: ParserF k a
  Branch :: k (Either x y) -> k (x -> a) -> k (y -> a) -> ParserF k a
```



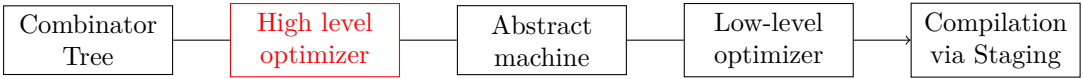

```
string :: String -> Parser String
string = traverse char

string ''ab''
-- urolling
pure (:) <*> char 'a' <*> (pure (:) <*> char 'b' <*> pure [])
-- Applicative fusion optimizations ....
satisfy (== 'a') *> satisfy (== 'b') *> pure ''ab''
```

Законы аппликативов

```
pure id <*> p = p  -- (1)
pure f <*> pure x = pure (f x)  -- (2)
u <*> pure x = pure (λf -> f x) <*> u -- (3)
u <*> (v <*> w) = pure (.) <*> u <*> v <*> w -- (4)
```

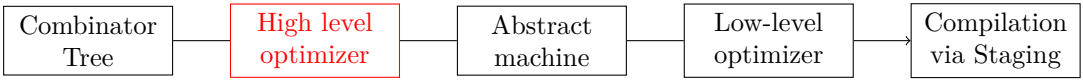
Вставка let




```
many :: Parser a → Parser a
many p = (p <:> many p) <▷ (pure p)

many :: Parser a → Parser a
many p =
  let go = (p <:> many p) <▷ (pure p)
  go
```

Различные анализы



- Cut
  - TODO
- Termination
  - Анализ точный для KC-грамматик и наивных аппликатиных парсеров
  - В остальных случаях анализ неточный и ложные срабатывания игнорируются

Сказать про CPS, и так как у нас нет монады и K3, то обходимся стековой машиной

```
data M (k :: [*] -> * -> * -> *) (xs :: [*]) (r :: *) (a :: *) where
  Halt :: M k [a] Void a
  Push :: x -> k (x : xs) r a -> M k xs r a
  Pop  :: k xs r a -> M k (x : xs) r a
  ...
```

- `k` is the same as in the combinator tree, it represents the shape of the values contained within each node (often Fix M);
- `xs` is a type-level list representing the types of the values required on the stack upon entry to the given instruction
- `r` represents what type the machine returns to the caller in the case that this is a recursive call
- `a` the final "goal" of the machine, in other words it directly corresponds to the type of the top-level parser that was compiled to generate this machine.

Компиляция (1/n)

```
compile :: Fix ParserF a -> Fix M [ ] Void a
compile = cata compAlg halt
type CodeGen a x = forall xs r . Fix M (x : xs) r a -> Fix M xs r a
compAlg :: ParserF (CodeGen a) x -> Fix M (x : xs) r a -> Fix M xs r a
compAlg (Pure x) = push x
compAlg (p :*>: q) = p . pop . q
compAlg (p :<*: q) = p . q . pop
```

Компиляция (2/n)

```
data M (k :: [*] -> * -> * -> *) (xs :: [*]) (r :: *) (a :: *) where
  ...
  Lift2 :: (x -> y -> z) -> k (z : xs) r a -> M k (y : x : xs) r a
  Swap :: k (x : y : xs) r a -> M k (y : x : xs) r a

app = lift2 id
compAlg (pf :<*>: px) = pf . px . app
```

Компиляция (1/n). Selectives

```
data M (k :: [*] -> * -> * -> *) (xs :: [*]) (r :: *) (a :: *) where
  ...
  Case :: k (x : xs) r a -> k (y : xs) r a -> M k (Either x y : xs) r a

compAlg (Branch b l r) = λk -> b (case (l (swap (app k))) (r (swap (app k))))
```

Компиляция (1/n). Alternatives

TODO: рассказать про две реализации alternative выше TODO: сказать про второй стек

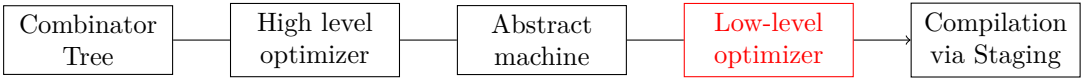
```
data M (k :: [*] -> * -> * -> *) (xs :: [*]) (r :: *) (a :: *) where
  ...
  Fail :: M k xs r a
  Catch :: k xs r a -> k (String : xs) r a -> M k xs r a
  Commit :: k xs r a -> M k xs r a

handle :: (Fix M xs r a -> Fix M (x : xs) r a) -> Fix M (String : xs) r a
  -> Fix M (x : xs) r a -> Fix M xs r a
handle p h k = catch (p (commit k)) h
compAlg (p :<|>: q) = λk -> handle p (parsecHandle (q k)) k
compAlg Empty = const fail
```

Компиляция (1/n). Primitive instructions

```
data M k (xs :: [*]) r a where
  ...
  Sat :: (Char -> Bool) -> k (Char : xs) r a -> M k xs r a
  Tell :: k (String : xs) r a -> M k xs r a
  Seek :: k xs r a -> M (String : xs) r a
  Ret :: M k [r] r a
  Call :: MuVar x -> k (x : xs) r a -> M k xs r a

compAlg (Satisfy p) = sat p
compAlg (Try p) = handle p (seek fail)
compAlg (Look p) = tell . p . swap . seek
compAlg (Let μ) = call μ
```



- Join points (φ-узлы)

  ```
  compAlg (p :<|>: q) = λk -> handle p (parsecHandle (q k)) k
  ```

  Решается с помощью специальной инструкции машины **Join**, которая вставляется после **Case** и **Catch**
- TCO
  - Достигается введением инструкций **Jump** и **Call**
- Deep inspection (via histomoprhism)

Consumption analysis revisited

Staging

Обычная функция возведения в степень

```
power :: Nat -> (Int -> Int)
power 0 = λx -> 1
power n = λx -> x * power (n - 1) x
```

Staged функция, где степень известна *статически*, а основание – динамически

```
power_ :: Nat -> Code (Int -> Int)
power_ 0 = [| λx -> 1 |]
power_ n = [| λx -> x * $(power_ (n-1)) |]
```

```
power5 = $(power_ 5)
       = $([| λx -> x * x * x * x * x * 1 |])
       = λx -> x * x * x * x * x * 1
```

Quoting vs. splicing

Если `x :: a` то `[|x|] :: Code a`

и если `qx :: Code a`, то `$(qx) :: a`.

Staging

```
type Eval' xs r a = Code (Γ xs r a -> Maybe a)
```

Заслуги работы

Оглавление

Результаты

Достижения:

I Спроектировали синтез с помощью комбинации *реляционных интерпретаторов* на miniKanren

II Заменили $\forall$ на *конечный* набор примеров

III Сделали оптимизацию методом ветвей и границ с помощью нового примитива miniKanren: *ограничение на структуру (structural constraint)*

Пути дальнейшего улучшения

- 1

Спасибо!

Литература