Implementacja języków EDSL w Haskellu

Piotr Krzemiński

Wrocław, 14 listopada 2013

Domain Specific Language – język dziedzinowy, wyspecjalizowany w konkretnej dziedzinie zastosowań

Cechy

- ograniczone środki abstrakcji
- semantyka dostosowana do konkretnej dziedziny
- nie musi być językiem ogólnego przeznaczenia
- mogą być używane przez ekspertów dziedzinowych, niekoniecznie będących programistami

Cechy

W językach DSL:

- programy są bardziej zwięzłe
- szybciej się je pisze
- są tańsze w utrzymaniu
- możemy łatwiej o nich wnioskować

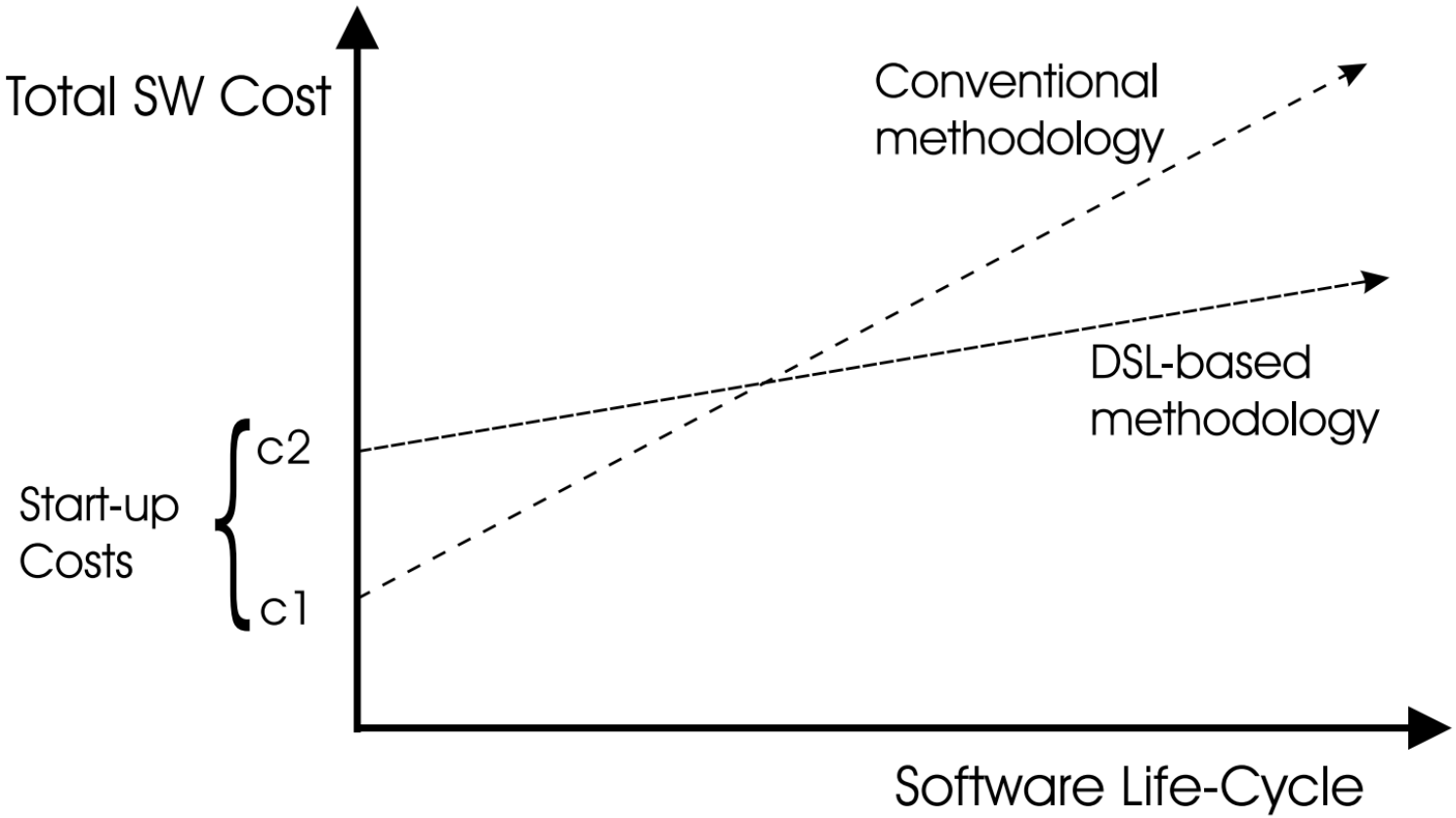
Szeroko używane języki dziedzinowe

- *OpenGL* – wysokopoziomowa grafika 3D
- *PostScript* – niskopoziomowa grafika
- *VHDL* – język opisu sprzętu
- *Lex, Yacc* – generatory lekserów, parserów
- *HTML* – język znaczników
- *LaTeX* – składanie dokumentów
- *Maple, Mathematica* – obliczenia symboliczne
- *DOT* – wizualizacja grafów
- *SQL* – zapytania do relacyjnych baz danych

Co musimy zrobić?

- wybrać precyzyjnie dziedzinę
- zaprojektować składnię języka, nadać jej znaczenie
- zaimplementować narzędzia (parser, interpreter, analizator, translator, debugger)

Koszt



źródło: Paul Hudak, *Modular Domain Specific Languages and Tools*

Teoria vs. rzeczywistość

- trudno zaprojektować i zaimplementować nowy język od zera
- może nam się nie udać za pierwszym razem
- duża szansa, że doświadczymy wszystkich bolączek ewolucji języka programowania...
- ...i tak możemy nie osiągnąć zamierzonego celu

Podejście Embedded

- nie chcemy budować naszego języka od zera
- wolelibyśmy wykorzystać infrastrukturę istniejącego języka programowania, tzw. języka-gospodarza (ang. *host language*)

Podejście Embedded

- nie chcemy budować naszego języka od zera
- wolelibyśmy wykorzystać infrastrukturę istniejącego języka programowania, tzw. języka-gospodarza (ang. *host language*)
- EDSL (DSEL) – *Embedded Domain Specific Language*

Podejście Embedded

- możemy skupić się na semantyce naszej dziedziny
- nie musimy implementować od zera narzędzi deweloperskich
- za darmo możemy korzystać z większości cech języka-gospodarza (np. składnia, funkcje, system typów)

EDSL – sposób implementacji

- metaprogramowanie + preprocessing (np. LISP-owe makra)
- *pure embedding*

Czy Haskell nadaje się do implementacji języków EDSL?

- elastyczna składnia
- klasy typów
- funkcje wyższego rzędu
- monady, transformatory monad
- silny system typów

Elastyczna składnia

- definiowane przez użytkownika operatory (i ich priorytety)
- przeciążanie (w szczególności przeciążanie literałów)
- notacja wywołań funkcji: bez nawiasów, infiksowa
- do-notacja
- rozszerzenia

Pozwala zbudować unikalny *look and feel*

Monady

- pełna kontrola nad efektami ubocznymi
- pozwalają ukryć stan obliczeń

Własności i prawa algebraiczne

- elementy neutralne
- operatory łączne
- operatory przemienne
- ...

Własności te możemy wyrazić, otypować, a często nawet automatycznie testować (np. korzystając z QuickChecka)

Implementując język EDSL w Haskellu, wyróżniamy dwa zasadnicze podejścia:

- płytka integracja (ang. *shallow embedding*)
- głęboka integracja (ang. *deep embedding*)

Płytka integracja – cechy

- konstrukcje języka reprezentowane bezpośrednio w semantyce
- względnie łatwo możemy korzystać z haskellowych mechanizmów (rekursja, współdzielenie, typy)
- zazwyczaj proste w implementacji
- przyzwoita wydajność
- mniejsza elastyczność
- jedyna słuszna interpretacja
- trudności przy debugowaniu/analizie

Głęboka integracja – cechy

- konstrukcje języka reprezentowane w postaci składni abstrakcyjnej, interpretowane w osobnym etapie
- dostępne różne możliwości interpretacji (ewaluacja, translacja do innego języka, np. maszynowego, pretty print)
- większa elastyczność
- implementacja może być bardziej lub mniej wydajna
- trudniejsze wykorzystanie haskellowego współdzielenia i rekursji

Analiza obszarów geometrycznych

```
type Point = ...
type Vector = ...
type Region = ...
inRegion :: Point -> Region -> Bool
distance :: Point -> Point -> Double
empty :: Region
circle :: Region
square :: Region
outside :: Region -> Region
scale :: Vector -> Region -> Region
translate :: Vector -> Region -> Region
(/\) :: Region -> Region -> Region
(\/) :: Region -> Region -> Region
```

Implementacja – płytka integracja

Obszar geometryczny będziemy reprezentować za pomocą funkcji charakterystycznej:

```
type Region = Point -> Bool
```

Aby sprawdzić czy punkt należy do obszaru, zdefiniujmy funkcję:

```
inRegion :: Point -> Region -> Bool
inRegion p r = r p
```

Implementacja – płytka integracja

```
inRegion p r = r p
distance (x1, y1) (x2, y2) =
  sqrt ((x1 - x2) ** 2 + (y1 - y2) ** 2)
empty _ = False
circle p = distance p (0,0) <= 1
square (x,y) = abs x <= 1 && abs y <= 1
outside r p = not (r p)
scale (sx, sy) r (x, y) = r (x / sx, y / sy)
translate (dx, dy) r (x, y) = r (x - dx, y - dy)
(r1 /\ r2) p = r1 p && r2 p
(r1 \/ r2) p = r1 p || r2 p
```

Implementacja – płytka integracja

Możemy wykorzystać haskellowe funkcje do zdefiniowania bardziej złożonych obszarów

```
circleAt :: Double -> Point -> Region
circleAt r center = translate center (scale (r, r) circle)

circleRow :: Integer -> Region
circleRow n = foldr (\/) empty
              [circleAt 1 (fromInteger x,0) | x <- [0..n-1]]
```

Implementacja – głęboka integracja

Teraz reprezentacja danych wygląda tak:

```
type Point = (Double, Double)
type Vector = (Double, Double)
data Region = Empty
            | Circle
            | Square
            | Scale Vector Region
            | Translate Vector Region
            | Outside Region
            | Intersect Region Region
            | Union Region Region
```

Implementacja – głęboka integracja

```
inRegion p Empty = False
inRegion p Circle = distance p (0,0) <= 1
inRegion (x, y) Square = abs x <= 1 && abs y <= 1
inRegion p (Outside r) = not (inRegion p r)
inRegion (x, y) (Scale (sx, sy) r) =
  inRegion (x / sx, y / sy) r
inRegion (x, y) (Translate (sx, sy) r) =
  inRegion (x - sx, y - sy) r
inRegion p (Intersect r1 r2) = inRegion p r1 && inRegion p r2
inRegion p (Union r1 r2) = inRegion p r1 || inRegion p r2
```

Implementacja – głęboka integracja

```
empty = Empty
circle = Circle
square = Square
outside = Outside
scale = Scale
translate = Translate
(/\) = Intersect
(\/) = Union
```

Własności geometryczne

. . .

Implementując języki EDSL w Haskellu, powinniśmy mieć świadomość, że w zależności od wybranego stopnia integracji mogą się pojawić problemy ze współdzieleniem obliczeń (ang. *sharing*) lub rekursją.

Prosty kalkulator

W celu zobrazowania problemów zdefiniujmy bardzo prosty język EDSL, o następującym interfejsie:

```
data Expr
one :: Expr
plus :: Expr -> Expr -> Expr
eval :: Expr -> Int
```

Jaki jest problem ze współdzieleniem?

Kalkulator w wersji płytkiej

```haskell
type Expr = Int
one = 1
plus = (+)
eval = id
```

Kalkulator w wersji płytkiej

```
*Main> one `plus` one `plus` one
3
*Main> eval (one `plus` one `plus` one)
3
```

Kalkulator w wersji płytkiej

Zdefiniujmy teraz prostą funkcję

```
treeI :: Int -> Expr
treeI 0 = one
treeI n = let shared = treeI (n-1) in shared `plus` shared
```

Kalkulator w wersji płytkiej

Zdefiniujmy teraz prostą funkcję

```
treeI :: Int -> Expr
treeI 0 = one
treeI n = let shared = treeI (n-1) in shared `plus` shared
```

Współdzielenie za pomocą Haskellowego `let`

Kalkulator w wersji płytkiej

Współdzielenie rzeczywiście działa w tym przypadku!

```
*Main> treeI 30
1073741824
```

Kalkulator w wersji płytkiej

Współdzielenie rzeczywiście działa w tym przypadku!

```
*Main> treeI 30
1073741824
```

Liczy się w ułamku sekundy, bo jest liniowe!

Kalkulator w wersji płytkiej

Niestety, płytka integracja jest za słaba, aby zdefiniować funkcję, która drukuje wyrażenie

```
text :: Expr -> String
text = ???
```

Kalkulator w wersji płytkiej

Niestety, płytka integracja jest za słaba, aby zdefiniować funkcję, która drukuje wyrażenie

```
text :: Expr -> String
text = ???
```

- płytka integracja ma swoje ograniczenia
- zbyt szybko zadaliśmy semantykę naszym termom

Kalkulator w wersji głębokiej

Spróbujmy zatem głębszej integracji

```
data Expr = One | Add Expr Expr

one = One
plus = Add

eval One = 1
eval (Add e1 e2) = eval e1 + eval e2
```

Kalkulator w wersji głębokiej

```
*Main> one `plus` one `plus` one
Add (Add One One) One
*Main> eval (one `plus` one `plus` one)
3
```

Kalkulator w wersji głębokiej

Teraz bez problemu możemy zdefiniować funkcję drukującą wyrażenia

```
text :: Expr -> String
text One = "1"
text (Add e1 e2) = "(" ++ text e1 ++ " + " ++ text e2 ++ ")"
```

Kalkulator w wersji głębokiej

Teraz bez problemu możemy zdefiniować funkcję drukującą wyrażenia

```
text :: Expr -> String
text One = "1"
text (Add e1 e2) = "(" ++ text e1 ++ " + " ++ text e2 ++ ")"

*Main> text (one `plus` one `plus` one)
"((1 + 1) + 1)"
```

Kalkulator w wersji głębokiej

Ale przy okazji popsuliśmy sobie współdzielenie :(

```
treeI :: Int -> Expr
treeI 0 = one
treeI n = let shared = treeI (n-1) in shared `plus` shared
```

Kalkulator w wersji głębokiej

Ale przy okazji popsuliśmy sobie współdzielenie :(

```
treeI :: Int -> Expr
treeI 0 = one
treeI n = let shared = treeI (n-1) in shared `plus` shared
```

```
*Main> eval (treeI 30)
```

Obliczenie trwa koszmarnie długo; czas wykładniczy

Co się stało?

Współdzielenie wprowadzone dzięki `let` jest tracone podczas `plus`

Winna jest reprezentacja danych

Remedium

Musimy jawnie zaimplementować współdzielenie

Drugie podejście

Dodajemy własną konstrukcję `let`

```
data Expr = One
          | Add Expr Expr
          | Let Expr (Expr -> Expr)

one = One
plus = Add
let_ = Let
```

Drugie podejście

Ale nadal mamy problem...

```
eval One = 1
eval (Add e1 e2) = eval e1 + eval e2
eval (Let e1 e2) = let shared = eval e1 in (e2 (??? shared))
```

Drugie podejście

Ale nadal mamy problem...

```
eval One = 1
eval (Add e1 e2) = eval e1 + eval e2
eval (Let e1 e2) = let shared = eval e1 in (e2 (??? shared))
```

- nie zgadzają nam się typy
- shared jest typu Int
- e2 jest typu Expr -> Expr

Co teraz?

Moglibyśmy „cytować” policzony term z powrotem do reprezentacji języka, ale takie podejście jest mało efektywne, tracimy zalety współdzielenia

Co teraz?

Jawnie zdefiniować w języku „opakowanie” dla policzonej wartości. Reprezentacja przestaje być drzewem, a zaczyna być grafem.

Bruno C. d. S. Oliveira, Andres Löh — *Abstract Syntax Graphs for Domain Specific Languages*

Jawne współdzielenie w ASG

Do reprezentacji grafu użyjemy PHOAS (ang. *parametric higher-order abstract syntax*)

```
data Expr a = One
            | Add (Expr a) (Expr a)
            | Var a
            | Let (Expr a) (a -> Expr a)

type ClosedExpr = forall a. Expr a

one = One
plus = Add
let_ :: Expr a -> (Expr a -> Expr a) -> Expr a
let_ e1 e2 = Let e1 (\x -> e2 (Var x))
```

Jawne współdzielenie w ASG

```haskell
eval :: Expr Int -> Int
eval One = 1
eval (Add e1 e2) = eval e1 + eval e2
eval (Var n) = n
eval (Let e1 e2) = let shared = eval e1 in eval (e2 shared)
```

Jawne współdzielenie w ASG

Wreszcie możemy zdefiniować naszą funkcję budującą term z wykorzystaniem jawnego współdzielenia.

```
treeE :: Int -> ClosedExpr
treeE 0 = one
treeE n = let_ (treeE (n - 1)) (\shrd -> shrd `plus` shrd)
```

Jawne współdzielenie w ASG

Wreszcie możemy zdefiniować naszą funkcję budującą term z wykorzystaniem jawnego współdzielenia.

```
treeE :: Int -> ClosedExpr
treeE 0 = one
treeE n = let_ (treeE (n - 1)) (\shrd -> shrd `plus` shrd)
```

Ale nadal możemy korzystać z Haskellowego `let`

```
treeI :: Int -> ClosedExpr
treeI 0 = one
treeI n = let shrd = treeI (n-1) in shrd `plus` shrd
```

Jawne współdzielenie w ASG

Takie współdzielenie jest również obserwowalne

```
text :: ClosedExpr -> String
text e = go e 0
 where
  go :: Expr String -> Int -> String
  go One _ = "1"
  go (Add e1 e2) c = "(" ++ go e1 c ++ " + " ++ go e2 c ++ ")"
  go (Var x) _ = x
  go (Let e1 e2) c =
   "(let " ++ v ++ " = " ++ go e1 (c + 1) ++
   " in " ++ go (e2 v) (c + 1) ++ ")"
    where
     v = "v" ++ show c
```

Jawne współdzielenie w ASG

```
*Main> text $ treeI 2
"((1 + 1) + (1 + 1))"
*Main> text $ treeE 2
"(let v0 = (let v1 = 1 in (v1 + v1)) in (v0 + v0))"
*Main> text $ treeI 3
"(((1 + 1) + (1 + 1)) + ((1 + 1) + (1 + 1)))"
*Main> text $ treeE 3
"(let v0 = (let v1 = (let v2 = 1 in (v2 + v2)) in (v1 + v1))
in (v0 + v0))"
```

Jawne współdzielenie w ASG

```
inline :: Expr (Expr a) -> Expr a
inline One = One
inline (Add e1 e2) = Add (inline e1) (inline e2)
inline (Var x) = x
inline (Let e1 e2) = inline (e2 (inline e1))
```

Jawne współdzielenie w ASG

```
*Main> text $ inline $ treeI 3
"(((1 + 1) + (1 + 1)) + ((1 + 1) + (1 + 1)))"
*Main> text $ inline $ treeE 3
"(((1 + 1) + (1 + 1)) + ((1 + 1) + (1 + 1)))"
```

Jawne współdzielenie w ASG

```
*Main> eval $ treeE 30 -- szybkie
*Main> eval $ treeI 30 -- powolne
*Main> eval $ inline $ treeE 30 -- powolne
```

Sharing – podsumowanie

- dwa rodzaje współdzielenia: *preservable* i *observable*
- przy płytkiej integracji wykorzystujemy haskellowy *preservable sharing* (`let`), nie możemy wyrazić *observable sharing*
- przy głębokiej integracji zazwyczaj sami musimy zaimplementować współdzielenie
- eleganckie rozwiązanie w postaci jawnego współdzielenia za pomocą ASG

Po co nam rekursja?

Wzbogacamy składnię

```haskell
data Expr a = Lit Int
            | Add (Expr a) (Expr a)
            | IfZero (Expr a) (Expr a) (Expr a)
            | Var a
            | Let (Expr a) (a -> Expr a)
            | Lam (a -> Expr a)
            | App (Expr a) (Expr a)

type ClosedExpr = forall a. Expr a
```

Sprytne konstruktory

```
plus = Add
apply = App
let_ e1 e2 = Let e1 (\x -> e2 (Var x))
lam_ e = Lam (\x -> e (Var x))
```

Funkcje jako wartości

```haskell
data Value = N Int | F (Value -> Value)
```

Ewaluator

```haskell
eval :: Expr Value -> Value
eval (Lit n) = N n
eval (Add e1 e2) = add (eval e1) (eval e2)
  where add (N m) (N n) = N (m + n)
eval (IfZero e1 e2 e3) = ifZero (eval e1) (eval e2) (eval e3)
  where ifZero (N n) v1 v2 = if n == 0 then v1 else v2
eval (Var x) = x
eval (Let e1 e2) = eval (e2 (eval e1))
eval (Lam e) = F (\v -> eval (e v))
eval (App e1 e2) = app (eval e1) (eval e2)
  where app (F f) v = f v
```

Przykład, jeszcze bez rekursji

Chcemy zakodować w naszym EDSL-u haskellowy term:

```
let dec x = x - 1
    twice f x = f (f x)
in  twice twice dec 10
```

Przykład, jeszcze bez rekursji

Chcemy zakodować w naszym EDSL-u haskellowy term:

```
let dec x = x - 1
    twice f x = f (f x)
in  twice twice dec 10
```

```
example1 = let_ (lam_ (\x -> x `plus` Lit (-1)))
  (\dec ->
    let_ (lam_ (\f -> lam_ (\x -> f `apply` (f `apply` x))))
     (\twice ->
       (twice `apply` twice `apply` dec `apply` (Lit 10))
     )
  )
```

Przykład, jeszcze bez rekursji

Chcemy zakodować w naszym EDSL-u haskellowy term:

```
let dec x = x - 1
    twice f x = f (f x)
in  twice twice dec 10
```

```
example1 = let_ (lam_ (\x -> x `plus` Lit (-1)))
  (\dec ->
    let_ (lam_ (\f -> lam_ (\x -> f `apply` (f `apply` x))))
     (\twice ->
       (twice `apply` twice `apply` dec `apply` (Lit 10))
     )
  )
```

```
*Main> eval example1
6
```

Dodajemy rekursję

```haskell
data Expr a = ... -- jak poprzednio
            | Mu (a -> Expr a)

mu_ e = Mu (\x -> e (Var x))

eval :: ClosedExpr -> Value
eval ... = ... -- jak poprzednio
eval (Mu e) = fix (\v -> eval (e v))
  where
    fix :: (a -> a) -> a
    fix f = let r = f r in r
```

Przykład z rekursją – mnożenie

Chcemy zakodować w naszym EDSL-u haskellowy term:

```
mul m = let rec n =
          if n == 0 then 0
          else m + (rec (n - 1))
        in rec
```

Przykład z rekursją – mnożenie

Chcemy zakodować w naszym EDSL-u haskellowy term:

```
mul m = let rec n =
          if n == 0 then 0
          else m + (rec (n - 1))
        in rec

mul :: ClosedExpr
mul = lam_ (\m -> mu_ (\rec -> lam_ (\n ->
        IfZero n (Lit 0)
        (m `plus` (rec `apply` (n `plus` Lit (-1))))
      )))
```

Przykład z rekursją – mnożenie

Chcemy zakodować w naszym EDSL-u haskellowy term:

```
mul m = let rec n =
          if n == 0 then 0
          else m + (rec (n - 1))
        in rec

mul :: ClosedExpr
mul = lam_ (\m -> mu_ (\rec -> lam_ (\n ->
        IfZero n (Lit 0)
        (m `plus` (rec `apply` (n `plus` Lit (-1))))
      )))

*Main> eval (mul `apply` (n 6) `apply` (n 3))
18
```

Przykład z rekursją – silnia

```
letrec_ :: (Expr a -> Expr a) -> (Expr a -> Expr a) -> Expr a
letrec_ e1 e2 = Let (Mu (\x -> e1 (Var x))) (\x -> e2 (Var x))
```

Przykład z rekursją – silnia

```
letrec_ :: (Expr a -> Expr a) -> (Expr a -> Expr a) -> Expr a
letrec_ e1 e2 = Let (Mu (\x -> e1 (Var x))) (\x -> e2 (Var x))

fact :: ClosedExpr
fact = letrec_ (\rec -> lam_ (\n ->
    IfZero n (Lit 1)
    (mul `apply` n `apply` (rec `apply` (n `plus` Lit (-1))))
  ))
  (\rec -> lam_ (\m -> rec `apply` m))
```

Przykład z rekursją – silnia

```
letrec_ :: (Expr a -> Expr a) -> (Expr a -> Expr a) -> Expr a
letrec_ e1 e2 = Let (Mu (\x -> e1 (Var x))) (\x -> e2 (Var x))

fact :: ClosedExpr
fact = letrec_ (\rec -> lam_ (\n ->
    IfZero n (Lit 1)
    (mul `apply` n `apply` (rec `apply` (n `plus` Lit (-1))))
  ))
  (\rec -> lam_ (\m -> rec `apply` m))

*Main> eval (fact `apply` (n 5))
120
```

Rekursja wzajemna

Tak zaimplementowana rekursja nie wystarczy jednak do zakodowania takiego termu:

```
let dec  x      = x - 1
    even x f e = if x == 0 then t else odd (dec x) t e
    odd  x f e = if x == 0 then e else even (dec x) t e
in even 4 1 0
```

Rekursja wzajemna

Tak zaimplementowana rekursja nie wystarczy jednak do zakodowania takiego termu:

```
let dec  x       = x - 1
    even x f e = if x == 0 then t else odd (dec x) t e
    odd  x f e = if x == 0 then e else even (dec x) t e
in even 4 1 0
```

Musimy jawnie reprezentować rekursję wzajemną!

Rekursja wzajemna

```haskell
data Expr a = ... -- jak poprzednio
            | LetRec ([a] -> [Expr a]) ([a] -> Expr a)

letrec_ :: ([Expr a]->[Expr a])->([Expr a]->Expr a)->Expr a
letrec_ es e = LetRec (\xs -> es (map Var xs))
                      (\xs -> e (map Var xs))

eval :: ClosedExpr -> Value
eval ... = ... -- jak poprzednio
eval (LetRec es e) = let rs = map eval (es rs) in eval (e rs)
```

Rekursja wzajemna

Teraz możemy zdefiniować już wspomniany term:

```
example = letrec_ (\ ~[dec,even,odd] ->
  [ lam_ (\x -> x `plus` Lit (-1))
   ,lam_ (\x -> lam_ (\t -> lam_ (\e ->
      IfZero x t
      (odd `apply` (dec `apply` x) `apply` t `apply` e)
   )))
   ,lam_ (\x -> lam_ (\t -> lam_ (\e ->
      IfZero x e
      (even `apply` (dec `apply` x) `apply` t `apply` e)
    )))
  ])
  (\ [dec,even,odd] ->
     even `apply` Lit 4 `apply` Lit 1 `apply` Lit 0)
```

Rekursja wzajemna

```
*Main> eval example
1
```

Cechy

- w naszym EDSL-u potrafimy wyrazić λ-abstrakcję, aplikację, jawne współdzielenie i (wzajemną) rekursję
- korzystając z haskellowych mechanizmów, nie musimy jawnie zajmować się środowiskami, domknięciami czy wiązaniem
- rekursja pozostaje obserwowalna! możemy napisać funkcję:

  ```
  text :: ClosedExpr -> String
  ```

  która wydrukuje nam nasze wyrażenie

Cechy

- możemy używać haskellowego `let` do współdzielenia,
- obserwowanie takiego współdzielenia nadal jest możliwe, ale wymaga sztuczki
- odbudowa naszego termu w jawnie reprezentowalny graf
- funkcja `reifyGraph` z pakietu `data-reify`

Co dalej?

- nasza implementacja rekursji wzajemnej operuje na dwu listach
- zmusza nas do użycia leniwego pattern matchingu w naszej reprezentacji
- wymaga to niejawnego niezmiennika, że rozmiar obu list musi być ten sam
- sprawniejsza implementacja: lista par

Co dalej?

- nasza reprezentacja nadal pozwala zdefiniować źle otypowane termy
- aby temu zaradzić, możemy wykorzystać system typów Haskella (aka. *well typed ASG*)

Idea

- wiele osób jedzie razem na wycieczkę
- nie zawsze każdy płaci za siebie
- pod koniec wycieczki chcemy się sprawnie rozliczyć

Zaczynamy od zdefiniowania osoby

```
newtype Person = Person { name :: String }
               deriving (Eq, Ord, Show)

dexter = Person "Dexter"
angel  = Person "Angel"
debra  = Person "Debra"
harry  = Person "Harry"
```

Chcemy, aby nasz DSL wyglądał tak:

```
trip = sharedexpenses $ do
    dexter `spent` 5300
    angel  `spent` 2700
    debra  `spent`  800
    harry  `spent` 1900
    debra  `spent` 1700
    angel  `spent` 2200
    dexter `gave` harry $ 2000
    angel  `gave` debra $ 3200
    angel  `gave` harry $ 500
```

Na koniec chcemy otrzymać minimalne rozwiązanie

```
*Main> solve trip
["Debra pays 4350 to Angel",
 "Harry pays 3650 to Dexter",
 "Harry pays 600 to Angel"]
```

Wydatki będziemy przechowywać w mapie opakowanej w monadę stanu.

```
sharedexpenses :: State (Map Person Int) () -> Map Person Int
sharedexpenses f = execState f empty

spent payer money = modify $ insertWith (+) payer money

gave lender borrower money = modify $
 (adjust (+ money) lender) . (adjust (subtract money) borrower)
```

```haskell
solve :: Map Person Int -> [String]
solve st = solve' err $ Map.map (\m -> m - avg) st
 where
  err = 1 + size st
  avg = round $ (toRational $ fold (+) 0 st) / (toRational $ size st)
solve' _ st | Map.null st = []
solve' err st =
 (name payer ++ " pays " ++ show amount ++ " to " ++ name receiver) : solve' err newstate
  where
   (payer, debt) = foldrWithKey (getpers True) (Person "", 0) st
   (receiver, credit) = foldrWithKey (getpers False) (Person "", 0) st
   getpers True p m (_, m0) | m < m0 = (p, m) -- Gets payer
   getpers False p m (_, m0) | m > m0 = (p, m) -- Gets receiver
   getpers _ _ _ e = e
   amount = min (-debt) credit
   newstate = Map.filter (\c -> c < -err || err < c) (mapWithKey statefix st)
   statefix p m | p == receiver = m - amount
   statefix p m | p == payer = m + amount
   statefix _ m = m
```

Działa!

```
*Main> solve $ sharedexpenses $ do
  angel  `spent`  300
  debra  `spent`  500
  dexter `gave` harry $ 200
  harry  `spent` 1000
  dexter `gave` debra $ 200
  dexter `spent`  600

["Debra pays 200 to Harry",
 "Angel pays 100 to Harry",
 "Angel pays 100 to Dexter"]
```

To wszystko!

- składnię naszego języka zaimplementowaliśmy za pomocą haskellowych funkcji
- do reprezentacji wykorzystaliśmy Haskellowe typy danych
- płytka integracja – obyło się bez parsera, AST, etc.

- Modular Domain Specific Language and Tools. Paul Hudak. Department of Computer Science, Yale University.
- Abstract Syntax Graphs for Domain Specific Languages. Bruno C. d. S. Oliveira, Andres Löh.
- Making EDSLs Fly, Lennart Augustsson, TechMesh 2012 http://www.youtube.com/watch?v=7gF7iFB4mFY
- DSL lectures, Veronica Gaspes http://www2.hh.se/staff/vero/dsl

Pytania?