Multi-staged programming

Radosław Warzocha

Motywacja

- chcemy pisać kod wielokrotnego użytku (reusable)
- dłuższy czas wykonania jest kosztem abstrakcji
- generatory kodu pozwalają na więcej

Motywacja

- chcemy pisać kod wielokrotnego użytku (reusable)
- dłuższy czas wykonania jest kosztem abstrakcji
- generatory kodu pozwalają na więcej

Multi-staged programming

To o co chodzi z tym MSP?

- rodzaj meta-programowania
- daje nam sposób na generowanie kodu w czasie kompilacji...
- ... jego częściową ewaluację...
- ... i jeszcze sprawdzenia poprawności typów

Generatory kodu

- reprezentacja jako łańcuchy znaków
- PROBLEM: poprawność syntaktyczna

```
"f(x, y)"  -> OK
"f(, y)"   -> ???
```

- reprezentacja jako typ danych (AST)
- PROBLEM: nie wiemy czy program jest poprawnie typowany

Generatory kodu

- reprezentacja jako łańcuchy znaków
- PROBLEM: poprawność syntaktyczna

```
"f(x, y)"  -> OK
"f(, y)"   -> ???
```

- reprezentacja jako typ danych (AST)
- PROBLEM: nie wiemy czy program jest poprawnie typowany

Konstrukcje

Nawiasy ($\langle \_ \rangle$)

Otoczenie fragmentu programu nawiasami oznacza, że chcemy je traktować jak kawałek kodu

```
1+2 :  int
⟨ 1+2 ⟩ :  int code
```

Konstrukcje

Escape (~_)

Otoczenie fragmentu programu nawiasami oznacza, że chcemy je traktować jak kawałek kodu

```
> x = ⟨ 1 + 2 ⟩
> y = ⟨ x + 3 ⟩
> y
⟨ (1 + 2) + 3 ⟩
```

Run

Taki fragment kodu możemy sobie wykonać

```
> x = ⟨ 1 + 2 ⟩
> run x
3
```

lift

Czasem będziemy jeszcze potrzebować ’podnieść’ wyrażenie do fragmentu kodu.

```
> lift 3
⟨ 3 ⟩
```

Konstrukcje

- nawiasy ($\langle \_ \rangle$) - opóźniają wykonanie instrukcji
- escape ($\tilde{}\_$) - służą do składania opóźnionych wartości
- `run` - generuje kod
- `lift` - 'podnosi' zwykłą wartość do typu kodu

Narzędzia

- MetaOCaml
- MetaML
- Template Haskell

Przykłady

Prosta funkcja obliczająca potęgę

```
power 0 x = 1
power n x = x * power (n-1) x

power2 = power 2

ext_power' 0 _ = lift (1 :: Integer)
ext_power' n x = [| $x * $(ext_power' (n-1) x) |]
power' n = [| \x -> $(ext_power' n [| x |]) |]

power2' = $(power' 2)
```

Przykłady

Prosta funkcja obliczająca potęgę

```
power 0 x = 1
power n x = x * power (n-1) x

power2 = power 2

ext_power' 0 _ = lift (1 :: Integer)
ext_power' n x = [| $x * $(ext_power' (n-1) x) |]
power' n = [| \x -> $(ext_power' n [| x |]) |]

power2' = $(power' 2)
```

Przykłady

A co gdybyśmy chcieli mieć `printf` jak w C?

```
> $(printf "Error: %s on line %d.") "Bad var" 123
"Error: Bad var on line 123."
```

Przykłady

```
printf :: String -> Q Exp
printf s = gen (parse s)

data Format = D | S | L String
parse :: String -> [Format]
-- parse "%d is %s" -> [D, L "is", S]
```

Przykłady

```
gen :: [Format] -> Q Exp
gen [D] = [| \n -> show n |]
gen [S] = [| \s -> s |]
gen [L s] = lift s
```

Przykłady

```
printf :: String -> Q Exp
printf s = gen (parse s) [| "" |]

gen :: [Format] -> Q Exp -> Q Exp
gen [] x = x
gen (D : xs) x = [| \n-> $(gen xs [| $x++show n |]) |]
gen (S : xs) x = [| \s-> $(gen xs [| $x++s |]) |]
gen (L s : xs) x = gen xs [| $x ++ $(lift s) |]
```

Bardziej zaawansowany przykład

Bardziej zaawansowany przykład: wybranie dow. el. z krotki

```
x = (a, b, c)
$(sel 1 3) x -- a

sel :: Int -> Int -> Q Exp
sel i n = [| \x -> case x of ??? |]
```

To się chyba nie uda...

Reprezentacja kodu

Musimy dowiedzieć się więcej o tym jak nasz kod jest reprezentowany

- `varx = VarE (newName "x")` reprezentuje wyrażenie `x`
- `patx = VarP (newName "x")` reprezentuje pattern `x`
- `str = LitE (StringL "str")` reprezentuje literał `"str"`
- `tuple = TupE [varx, str]` reprezentuje krotkę `(x,"str")`
- `LamE [patx] tuple` reprezentuje lambdę `(\x -> (x,"str"))`

jest jeszcze wiele innych.

Reprezentacja kodu

Musimy dowiedzieć się więcej o tym jak nasz kod jest reprezentowany

- `varx = VarE (newName "x")` reprezentuje wyrażenie `x`
- `patx = VarP (newName "x")` reprezentuje pattern `x`
- `str = LitE (StringL "str")` reprezentuje literał `"str"`
- `tuple = TupE [varx, str]` reprezentuje krotkę `(x,"str")`
- `LamE [patx] tuple` reprezentuje lambdę `(\x -> (x,"str"))`

jest jeszcze wiele innych.

Bardziej zaawansowany przykład

```
sel :: Int -> Int -> Q Exp
sel i n = do id <- newName "x"
 return $ LamE [VarP id] (CaseE (VarE id) [alt])
 where  alt :: Match
        alt = Match pat (NormalB rhs) []

        pat :: Pat
        pat = TupP $ map (VarP . mkName) as

        rhs :: Exp
        rhs = VarE $ mkName (as !! (i-1))

        as :: [String]
        as = ["a" ++ show i | i <- [1..n]]
```

Bardziej zaawansowany przykład

Możemy też wymieszać oba podejścia

```
sel :: Int -> Int -> Q Exp
sel i n = [| \x -> $(caseE [| x |] [alt] |]
 where alt = match pat (normalB rhs) []

       pat = tupP $ map varP as

       rhs = varE (as !! (i-1))

       as = map mkName ["a" ++ show i | i <- [1..n]]
```

Co jeszcze możemy zrobić z TH

- deriving

```
data T a = Tip a | Fork (T a) (T a)
$(genEq (reifyDecl T))
```

- definiowanie wielu funkcji naraz

```
$(genZips 20)
```

- mamy pewność, że wszystko jest porządnie typowane...
- i że nie wystąpią konflikty nazw

```
cross2e f g =
do  (vf,p) <- genpat [p| (x,y) |]
lamE [p] [| ( $f $(vf "x"), $g $(vf "y") ) |]
```