

Lista zadań nr 8

Zadanie 1. (2 pkt)

Jedną z optymalizacji, którą może przeprowadzić kompilator, jest *propagacja stałych*. Jeśli w programie napiszemy wyrażenie postaci $20 + 30$, kompilator nie wyprodukuje kodu, który oblicza $20 + 30$, tylko wyliczy wartość 50 w trakcie kompilacji. Jednym ze sposobów implementacji tej optymalizacji jest transformacja programu (w składni abstrakcyjnej) do uproszczonego, ale równoważnego programu (w składni abstrakcyjnej). Przykładowo, chcielibyśmy program

```
Binop (Mult, Var "x", Binop (Add, Int 20, Int 30))
```

przekształcić w program

```
Binop (Mult, Var "x", Int 50)
```

Dla nadania zadaniu realizmu, zakładamy, że program wejściowy może zawierać zmienne wolne: nie można więc zaimplementować go dokonując ewaluacji całości do pojedynczej wartości.

Prócz wyrażeń arytmetycznych, optymalizacja powinna być w stanie uprościć wyrażenia boolowskie, wyrażenia warunkowe – np.

```
If (Binop (Eq, Int 4, Binop (Add, Int 2, Int 2)), Int 1, Int 0)
```

do

```
Int 1
```

– oraz let-wyrażenia – np.

```
Let ("x", Binop (Add, Int 2, Int 3), Binop (Add, Var "x", Var "y"))
```

do

```
Binop (Add, Int 5, Var "y")
```

Zaimplementuj funkcję

```
cp : expr -> expr
```

przeprowadzającą propagację stałych dla języka LET z wykładu (dla uproszczenia: bez leniwych operacji `&&` i `||`).

Bardzo ważna wskazówka: Funkcję `cp` najlepiej zaimplementować jako rodzaj ewaluatora, w którym wartościami są wyrażenia. Przykładowo, wynikiem ewaluacji programu

```
Binop (Add, Var "x", Int 2)
```

jest program

```
Binop (Add, Var "x", Int 2)
```

ale wynikiem ewaluacji programu

```
Binop (Add, Int 3, Int 2)
```

jest program

```
Int 5
```

Każda konstrukcja w ewaluatorze musi więc odpowiednio przeanalizować wynik uproszczenia dla swoich podwyrażeń. Podobnie jak w przypadku zwykłej ewaluacji, środowisko powinno przechowywać wyniki ewaluacji (wyrażenia!) let-wyrażeń. W ten sposób, w wyrażeniu `Let ("x", e1, e2)` – o ile wyrażenie `e1` upraszcza się do stałej – w wyrażeniu `e2` możemy upraszczać wyrażenia zawierające zmienną `x`.

Zadanie 2. (2 pkt)

Dwa wyrażenia nazywamy α -równoważnymi, gdy różnią się tylko nazwami zmiennych związanych i mają taką samą strukturę przykrywania zmiennych. Przykładowo, wyrażenia w składni konkretnej

```
let x = 2 in let y = 5 in x + y
let y = 2 in let z = 5 in y + z
```

```
let x = 2 in x + y
let z = 2 in z + y
```

są parami α -równoważne, a

```
let x = 2 in let y = 5 in x + y
let y = 2 in let y = 5 in y + y
```

```
let x = 2 in x + y
let y = 2 in y + y
```

nie są.

Zaimplementuj funkcję

```
alpha_equiv : expr -> expr -> bool
```

która sprawdza czy dwa wyrażenia dla języka LET z wykładu są α -równoważne.

Zadanie 3. (2 pkt)

Dla języka LET z wykładu zaimplementuj funkcję

```
rename_expr : expr -> expr
```

która przekształca wyrażenie w α -równoważne wyrażenie, w którym nazwy wszystkich zmiennych związanych są różne. Przykładowo, wyrażenie dane w składni konkretnej jako

```
let x = 1 in
  (let y = 2 in x + y + z) + (let x = x in x)
```

może zostać przekształcone na wyrażenie

```
let v1 = 1 in
  (let v2 = 2 in v1 + v2 + z) + (let v3 = v1 in v3)
```

Uwaga: Dla uproszczenia, żeby uniknąć przykrycia zmienną związaną zmiennej wolnej, dla nowych nazw zmiennych możesz użyć symboli, które nie są dozwolone w składni konkretnej, i reprezentować powyższe wyrażenie w składni abstrakcyjnej np. jako:

```
Let ("#1", Int 1,
    Binop (Add, Let ("#2", Int 2,
                    Binop (Add, Var "#1" + Binop (Add, Var "#2", Var "z"),
                    (Let "#3", Var "#1", Var "#3"))
```

Zadanie 4. (2 pkt)

Do języka FUN z wykładu dodaj lukier syntaktyczny umożliwiający definiowanie funkcji wieloargumentowych przez łańcuszek definicji funkcji jednoargumentowych. Znaczy to, że parser dla wyrażenia postaci

```
fun x y z -> x + z
```

powinien produkować następujące wyrażenie w składni abstrakcyjnej:

```
Fun ("x",
    Fun ("y",
        Fun ("z",
            Binop (Add, Var "x", Var "Z"))))
```

Rozszerz odpowiednio gramatykę składni konkretnej i akcje semantyczne. Kod pomocniczy w parserze można umieścić w sekcji `%{ ... %}`:

```
%{
open Ast

(* Tutaj można wstawić funkcje pomocnicze *)
```

```
%}
```

Zadanie 5. (2 pkt)

Rozszerz parser języka FUN z wykładu o lukier syntaktyczny `let rec`, dzięki któremu programista zamiast pisać

```
let f = fix (fun f -> fun x -> e) in ...
```

może napisać

```
let rec f x = e in ...
```

Zadanie 6. (2 pkt)

Przypomnijmy, że w języku FUN natrafiliśmy na problem przy próbie zdefiniowania funkcji rekurencyjnych. Wartością wyrażenia

```
fun x -> e
```

w środowisku `env` jest domknięcie

```
VClosure (x, e, env)
```

Gdybyśmy chcieli w ten sam sposób potraktować funkcje rekurencyjne

```
let rec f x = e in ...
```

to musielibyśmy dołożyć do środowiska zapisanego w domknięciu także wartość dla zmiennej `f` (która widoczna jest w wyrażeniu `e`), którą to wartością jest właśnie to domknięcie, co powoduje zapętlenie nie do rozwiązania w znanej nam części OCaml'a.

Alternatywnym sposobem implementacji funkcji rekurencyjnych są domknięcia z *późnym wiązaniem*, w których nie zapamiętujemy wartości zmiennej `f` w momencie tworzenia, a w momencie wywołania funkcji¹. Takie domknięcie nie oczekuje więc już tylko zmiennej `x`, ale także zmiennej `f`. Podczas ewaluacji aplikacji, jeśli wyrażenie z lewej strony obliczy się do domknięcia z późnym wiązaniem, obliczamy wartość ciała funkcji zapisanego w domknięciu w środowisku z domknięcia, które uzupełniamy o wartość argumentu (czyli o wynik ewaluacji prawego argumentu aplikacji) i o wartość zmiennej „funkcyjnej” (czyli o nasze domknięcie), unikając rekurencyjnej zależności w samym domknięciu.

Rozszerzając składnię abstrakcyjną o rekurencyjne `let-y`:

¹Proszę przypomnieć sobie naszą obserwację z wykładu, że wyrażenia `fun x -> e` obliczają się od razu do wartości, więc brak zmiennej `f` w domknięciu może zostać zaobserwowany i tak dopiero w momencie wywołania funkcji!

```
type expr =  
  | Letrec of ident * ident * expr * expr  
  | ...
```

a wartości o domknięcia z późnym wiązaniem:

```
type env = ...  
and value =  
  | VRClosure of ident * ident * expr * env  
  | ...
```

rozszerz parser i ewaluator zgodnie z powyższym nieformalnym opisem.