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 do 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.

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`.

Wskazówka: Przyjrzyjmy się kilku przypadkom algorytmu propagacji stałych w środowisku `env`, które przypisuje wyrażeniom wyrażenia:

- Jeśli wyrażenie jest arytmetycznym operatorem binarnym $\oplus$, uprość lewe i prawe podwyrażenie.
  - Jeśli podwyrażenia uproszczą się do wyrażeń o kształacie `Int a` oraz `Int b`, wynikiem jest wyrażenie `Int (a` $\oplus$ `b)`.
  - W przeciwnym przypadku, wynikiem uproszczenia jest `Binop (`$\oplus$`, x, y)`, gdzie `x` i `y` to wyniki uproszczeń lewego i prawego podwyrażenia.
- Wynikiem uproszczenia zmiennej jest wyrażenie przypisane jej w środowisku. Jeśli środowisko nie przypisuje żadnej wartości zmiennej, wynikiem uproszczenia jest ta zmienna.
- Żeby uprościć wyrażenie `Let(x, e1, e2)`, najpierw upraszczamy wyrażenie `e1`:
  - Jeśli `e1` uprościło się do wyrażenia `Int a` lub `Bool b`, wynikiem uproszczenia jest wyrażenie `e2` uproszczone w środowisku, które rozszerzamy o przypisanie zmiennej `x` wyrażenia, odpowiednio, `Int a` lub `Bool b`.
  - Jeśli `e1` uprościło się do jakiegoś innego wyrażenia `e1'`, wynikiem uproszczenia jest `Let (x, e1', e2')`, gdzie `e2'` jest wynikiem uproszczenia wyrażenia `e2` w środowisku, któremu zmiennej `x` przypisujemy wyrażenie `Var x` (czemu nie `e1'`?).

Czy widzisz, że algorytm ma strukturę bardzo podobną do ewaluacji wyrażenia do wartości?

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.

Wskazówka: Zadanie to można rozwiązać na kilka sposobów.

Sposób 1: Pierwszy sposób to zaimplementować funkcję rekurencyjnie porównującą wyrażenia `e1` i `e2`, która dodatkowo posiada dwa środowiska: jedno odwzorowuje zmienne związane wyrażenia `e1` na zmienne (!) związane wyrażenia `e2`, a drugie odwrotnie – zmienne wyrażenia `e2` na zmienne wyrażenia `e1`. Przykładowo, żeby sprawdzić czy wyrażenia `Let ("x1", e1l, e1r)` oraz `Let ("x2", e2l, e2r)` w środowiskach `env1` i `env2` są α-równoważne, wystarczy sprawdzić czy `e1l` i `e2l` są równoważne w tych samych środowiskach oraz czy `e1r` i `e2r` są równoważne w środowiskach `M.add "x1" "x2" env1` i `M.add "x2" "x1" env2`. Porównując wyrażenia będące zmiennymi, sprawdzamy czy ich wartości w środowiskach wzajemnie sobie odpowiadają. Jeśli zmiennej nie ma w środowisku, oznacza to, że jest to zmienna wolna w całym wyrażeniu. Czy rozumiesz po co potrzebne są aż dwa środowiska?

Sposób 2: Przekształcić wyrażenie na reprezentację wykorzystującą *indeksy de Bruijna*. W takiej reprezentacji zmienne związane w ogóle nie mają nazw, a ich wystąpienia oznaczone są liczbami naturalnymi, które mówią, ile wiązań należy pominąć na ścieżce od zmiennej do korzenia, by dojść do odpowiadającego wiązania. Przykładowo, wyrażenie

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

można przedstawić jako

```
let 2 in (var 0) + let (var 0) + 1 in (var 1) + (var 0) + z
```

W takiej reprezentacji dwa termy są α-równoważne wtedy i tylko wtedy, gdy są równe. Jak przekształcić wyrażenie do postaci de Bruijna? Użyć rekurencyjnej funkcji ze środowiskiem, które nie jest słownikiem, a stosem: wiązanie wrzuca nazwę zmiennej na stos, a wystąpienie zmiennej zastępujemy indeksem na stosie. Jeśli danej zmiennej nie ma na stosie, znaczy to, że w wyrażeniu wejściowym była wolna.

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")
```

Wskazówka: Implementacja może być funkcją rekurencyjną ze środowiskiem, które przypisuje zmiennym nowe nazwy zeminnych. Nowe (w slangu mówimy „świeże”) nazwy powinny być generowane oczywiście w przypadku, gdy wyrażenie jest wiązaniem (czyli let-em). Jak generować świeże identyfikatory? Nie wystarczy mieć licznika przekazywanego w głąb rekursji, gdyż nie zagwarantuje to unikalności nazw w wyrażeniach typu

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

Możliwym rozwiązaniem jest przekazywanie w głąb rekursji kodowania ścieżki prowadzącej od korzenia wyrażenia, wówczas dostalibyśmy reprezentację w stylu:

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

Do generowania globalnie świeżych zmiennych dałoby się też użyć mutowalnej komórki pamięci (ale nie wolno) albo monady stanowej (ale o tym na innym przedmiocie).

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 mozna wstawic funkcje pomocnicze *)

%}
```

Zadanie 5. (2 pkt)

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

```
let f = funrec f -> fun x -> e in ... f ...
```

może napisać

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

Zadanie 6. (1 pkt)

Rozwiąż w języku FUN zadanie 3. z listy 2.

Uwaga: Kostrukcję `mod` możesz dodać jako nową konstrukcję do języka albo zdefiniować jako funkcję pomocniczą w języku, korzystając z faktu, że mamy już w nim mnożenie, odejmowanie i dzielenie całkowitoliczbowe.

Zadanie 7. (1 pkt)

Interesującą alternatywą dla rekurencyjnych domknięć jest skorzystanie z *kombinatora punktu stałego*. Przykładowo, kombinator Y możemy zdefiniować w języku FUN jako:

```
fun f -> (fun x -> fun y -> f (x x) y) (fun x -> fun y -> f (x x) y)
```

By zdefiniować funkcję rekurencyjną, należy zastąpić rekrencyjne wywołania wywołaniami dodatkowego argumentu, a następnie taką funkcją nakarmić kombinator, który „zawiąże węzeł” aplikując taką funkcję do samej siebie. Przykładowo, funkcję `fib` możemy zdefiniować następująco:

```
let fib =
  let fix = fun f -> (fun x -> fun y -> f (x x) y)
                     (fun x -> fun y -> f (x x) y) in
  let FIB = fun f -> fun x -> if x <= 1 then x
                                         else f (x - 1) + f (x - 2) in
  fun n -> fix FIB n
```

Jak widać, zwykłe domknięcia wystarczają do zbudowania funkcji rekurencyjnych, bez potrzeby posiadania osobnej konstrukcji dla domknięć rekurencyjnych.

Dodaj do języka FUN lukier syntaktyczny `funfix f x -> e` (będący alternatywą dla `funrec`), który rozwijany jest w parserze do funkcji o kształcie podobym do powyższego.

Co się stanie, jeśli spróbujemy zdefiniować kombinator Y w OCamlu?