Lista zadań nr 12

Poniższe zadania rozwiąż w języku Plait.

Zadanie 1. (2 pkt)

Zmodyfikuj interpreter z pliku `lambda-clo.rkt` tak, by aplikacja funkcji była leniwa, tzn., by argument funkcji był obliczany nie w trakcie aplikacji, a dopiero wtedy, gdy jego wartość będzie potrzebna, być może wielokrotnie. Reprezentacja wartości zwracanych przez ewaluator się nie zmieni (przynajmniej na pierwszy rzut oka), ale w środowisku będziemy przechowywać odroczone obliczenia, a nie wartości. Podobnie należy obsłużyć ewaluację `let`-wyrażeń.

Zademonstruj na przykładzie różnicę w działaniu Twojego ewaluatora i tego gorliwego z wykładu.

Zadanie 2. (2 pkt)

Zmień reprezentację zmiennych i ich wiązania w języku z funkcjami (`lambda-clo.rkt`) na adresowanie leksykalne, zgodnie ze schematem z pliku `let-lex-addr.rkt` z poprzedniego tygodnia. Idea pozostaje ta sama: liczba naturalna reprezentująca zmienną oznacza liczbę konstrukcji wiążących (`lambda` i `let`), które trzeba przejść od danego wystąpienia zmiennej do miejsca jej związania, np. wyrażenie

```
{lambda {x} {let y {* x x} {+ x y}}}
```

ma następującą reprezentację bez nazw

```
(lambdaA
  (letA (opA (mul) (varA 0) (varA 0))
    (opA (add) (varA 1) (varA 0))))
```

Wystarczy rozszerzyć kod z pliku `let-lex-addr.rkt` o obsługę funkcji i aplikacji.

Wyjaśnij na przykładzie dlaczego taka reprezentacja z indeksami dobrze współgra z domknięciami funkcji.

Zadanie 3. (2 pkt)

Zmodyfikuj interpreter z pliku `lambda-clo-primop.rkt`, tak by implementował wiązanie dynamiczne zamiast statycznego. W tym celu wystarczy zamiast budować domknięcia funkcji, przekazać funkcji `apply` środowisko. Na przykład, ewaluacja wyrażenia

```
{let a 1
  {let p {lambda {x} {+ x a}}
    {let a 10
      {p 0}}}}
```

przy wiązaniu dynamicznym powinna dać wartość 10, podczas gdy przy wiązaniu statycznym wartością tego wyrażenia jest 1.

Na powyższym przykładzie pokaż, że przy wiązaniu dynamicznym nazwy zmiennych związanych mają znaczenie.

Czy w języku z wiązaniem dynamicznym potrzebujemy konstrukcji `letrec`? A może każdy `let` to `letrec` i silnię w takim języku można napisać tak

```
{let fact {lambda {n}
            {if {= n 0}
                1
                {* n {fact {- n 1}}}}}}
  {fact 5}}
```

?

Zadanie 4. (2 pkt)

Operator punktu stałego `fix`, jak widzieliśmy na wykładzie (plik `recursion.rkt`), może zostać wyrażony w składni konkretnej języka z pliku `lambda-clo-primop.rkt`, np. tak:

```
  {lambda {f}
    {let w {lambda {g} {f {lambda {z} {{g g} z}}}}
      {w w}}}
```

gdzie parametr `f` reprezentuje funkcjonał opisujący definicję rekurencyjną, np. `f_fact`:

```
  {lambda {fact}
    {lambda {n}
      {if {= n 0}
          1
          {* n {fact {- n 1}}}}}}
```

Wówczas

```
  {fix f_fact}
```

reprezentuje funkcję silni.

Rozszerz język z pliku `lambda-clo-primop.rkt` o wyrażenie `letrec` wykorzystując operator punktu stałego. Wystarczy odpowiednio rozszerzyć parser. Czy w tym języku da się napisać funkcję wyznaczającą n-ty wyraz ciągu Fibonacciego? A jak Twój interpreter zareaguje na wyrażenie

```
  {letrec x {+ x 1} x}
```

?

Zadanie 5. (2 pkt)

Rozszerz język z pliku `lambda-clo-primop.rkt` i jego interpreter o funkcje rekurencyjne postaci `(rec f (x) e)`, gdzie `x` jest parametrem formalnym funkcji, a `f` jest widoczna w ciele funkcji `e` i reprezentuje właśnie definiowaną funkcję. Np. funkcja reprezentująca silnię może być w takim języku wyrażona następująco:

```
{rec fact {n} {if {= n 0} 1 {* n {fact {- n 1}}}}}
```

Ewaluacja takiego wyrażenia powinna dawać nowy rodzaj wartości (domknięcia). Funkcja `apply` zajmie się aplikacją takich domknięć (ona będzie wiedziała, że ma do czynienia z funkcją rekurencyjną i że trzeba odpowiednio rozszerzyć środowisko).

Wyjaśnij dlaczego `letrec` w ogólnej postaci, omówionej na wykładzie, nie mógł być potraktowany analogicznie.

Zadanie 6. (2 pkt)

Rozszerz język z pliku `letrec-state.rkt` i jego interpreter, o konstrukcję

```
{letmutrec {x1 e1} {x2 e2} e}
```

która umożliwia wzajemnie rekurencyjną definicję `x1` i `x2`, tak jak w poniższym przykładzie

```
{let true {= 0 0}
  {let false {= 0 1}
    {letmutrec
      {even {lambda {n} {if {= n 0} true {odd {- n 1}}}}}
      {odd {lambda {n} {if {= n 0} false {even {- n 1}}}}}
      {even 6}}}}
```

Czy potrafisz uogólnić swoje rozwiązanie na `letrec` z dowolną liczbą wiązań?