Lista zadań nr 7

Programowanie z typami

Poniższe dwa zadania rozwiąż w języku Plait.

Zadanie 1. (2 pkt)

Wstawianie do zwykłych binarnych drzew przeszukiwań może prowadzić do powstawania bardzo długich ścieżek w drzewie, a co za tym idzie do powolnego działania operacji na takim drzewie. Dlatego zaproponowano wiele struktur danych bazujących na drzewach przeszukiwań, ale dla których operacje wstawiania i usuwania dbają o to by drzewo było (w miarę) zbalansowane. Jedną z takich struktur są 2-3 drzewa, które można opisać za pomocą następujących warunków:

- drzewo składa się z liści oraz wierzchołków dwóch rodzajów: takich co mają jeden element i dwoje dzieci oraz takich co mają dwa elementy i troje dzieci;
- wszystkie ścieżki od korzenia do liścia mają tę samą długość, zwaną *wysokością* drzewa;
- dla każdego wierzchołka przechowywany (każdy) element jest większy od wszystkich elementów lewego poddrzewa;
- dla każdego wierzchołka przechowywany (każdy(element jest mniejszy od wszystkich elementów prawego poddrzewa;
- dla wierzchołków mających troje dzieci lewy element (a) jest większy od prawego (b), a wszystkie elementy środkowego poddrzewa są pomiędzy a i b.

Zdefiniuj typ opisujący 2-3 drzewa. Które własności 2-3 drzew potrafisz wymusić samą definicją typu? Napisz predykat sprawdzający pozostałe warunki.

Zadanie 2. (2 pkt)

Operacja wstawiania do 2-3 drzew albo nie zmienia jego wysokości, albo ją zwiększa o jeden. Jednak w tym drugim przypadku korzeń drzewa wynikowego zawsze ma dwoje

dzieci. Tą własność wstawiania można wyrazić w typie rekurencyjnej funkcji pomocniczej realizującej wstawianie do 2-3 drzewa. Zdefiniuj odpowiednie typy pomocnicze i podaj jaki może być typ tej pomocniczej funkcji (nie musisz jej definiować, ale jak chcesz, to możesz spróbować). Zakładając, że masz taką funkcję zdefiniowaną, napisz właściwą (nierekurencyjną) funkcję `insert`.

Kontrakty

Poniższe zadania dotyczą mechanizmu kontraktów. Należy je rozwiązać przy użyciu języka Racket.

Zadanie 3.

Napisz procedurę `suffixes`, zwracającą wszystkie sufiksy listy podanej jako argument. Napisz dla tej procedury odpowiedni kontrakt parametryczny.

Zadanie 4. (2 pkt)

Poniższa procedura ma za zadanie obliczyć listę wszystkich podlist listy podanej jako argument:

```
(define (sublists xs)
  (if (null? xs)
      (list null)
      (append-map
         (lambda (ys) (cons (cons (car xs) ys) ys))
         (sublists (cdr xs)))))
```

Niestety, procedura ta zawiera błąd:

```
> (sublists '(1 2))
'((1 2) 2)
> (sublists '(1 2 3))
'((1 2 3) 2 3 (1 . 3) . 3)
```

Napisz kontrakt parametryczny dla tej procedury, który odrzuci błędne wyniki. Popraw procedurę, aby działała zgodnie z założeniem oraz spełniała swój kontrakt.

Zadanie 5.

Wskaż w poniższych kontraktach wystąpienia pozytywne i negatywne. Zaimplementuj procedury spełniające te kontrakty.

```
(parametric->/c [a b] (-> a b a))
(parametric->/c [a b c] (-> (-> a b c) (-> a b) a c))
```

```
(parametric->/c [a b c] (-> (-> b c) (-> a b) (-> a c)))
(parametric->/c [a] (-> (-> (-> a a) a) a))
```

Zadanie 6. (2 pkt)

Poniższy kod implementuje procedurę łączącą w sobie cechy `foldl` i `map`:

```
(define (foldl-map f a xs)
  (define (it a xs ys)
    (if (null? xs)
        (cons (reverse ys) a)
        (let [(p (f (car xs) a))]
          (it (cdr p)
              (cdr xs)
              (cons (car p) ys)))))
  (it a xs null))
```

Pierwszy argument powinien być procedurą przyjmującą dwa argumenty, oznaczające (w kolejności) bieżący element listy oraz bieżący akumulator, zaś zwracającą parę złożoną z nowego elementu listy oraz nowej wartości akumulatora. Pozostałe dwa argumenty powinny zawierać startową wartość akumulatora oraz listę elementów do przetworzenia. Procedura `foldl-map` zwraca parę złożoną z listy wynikowej i końcowej wartości akumulatora.

Przykładowe wywołanie procedury, obliczające sumy częściowe:

```
(foldl-map (lambda (x a) (cons a (+ a x))) 0 '(1 2 3))
```

Napisz kontrakt parametryczny dla tej definicji. Zastosuj w kontrakcie jak najwięcej (prawidłowo użytych) parametrów.

Zadanie 7. (2 pkt)

Funkcji `fold-right` możemy nadać następujący kontrakt parametryczny:

```
(parametric->/c [a b] (-> (-> a b b) b (listof a) b))
```

W języku Plait ta funkcja otrzymuje następujący, analogiczny do powyższego kontraktu typ parametryczny:

```
(('a 'b -> 'b) 'b (Listof 'a) -> 'b)
```

Możemy rozważyć zmienione wersje kontraktu i typu powyżej, gdzie zamiast dwóch parametrów `a` i `b` użyjemy tylko jednego, `a`, który zastąpi wszystkie wystąpienia `a` i `b`. Odpowiedz na pytania:

- Jaka błędna implementacja procedury `fold-right` będzie spełniać zmienioną wersję kontraktu i mieć zmienioną wersję typu, a zostanie odrzucona przez wersje oryginalne?

Uwaga: z powodu nietypowego zachowania interpretera Plaita, aby sprawdzić, czy procedura `fold-right` ma powyższy typ, należy napisać w REPLu:

```
(has-type foldr-right : (('a 'b -> 'b) 'b (Listof 'a) -> 'b))
```

a następnie zwrócić uwagę, czy typ wypisany przez REPLa jest tym, którego żądaliśmy.

- Czy zmieniona wersja kontraktu ogranicza sposób użytkowania procedury? A zmieniona wersja typu?