MP20 @ II UWr 18 maja 2020 r.

Lista zagadnień nr 12

Kontrakty i komponenty

Ćwiczenie 1.

Zaproponuj kontrakt zależny (tzn. wykorzystujący `->i`) sprawdzający poprawność wyniku funkcji `sqrt` z wykładu.

Ćwiczenie 2.

Napisz kontrakt parametryczny dla funkcji `filter`. Zaproponuj rozszerzenie go do kontraktu zależnego, sprawdzającego wybraną własność tej funkcji. Nie przejmuj się efektywnością kontraktu.

Ćwiczenie 3.

Poniższa sygnatura opisuje *monoid* – strukturę algebraiczną znaną też jako półgrupa z elementem neutralnym:

```
(define-signature monoid^
  ((contracted
    [elem?   (-> any/c boolean?)]
    [neutral elem?]
    [oper    (-> elem? elem? elem?)])))
```

Zdefiniuj dwa komponenty implementujące tę sygnaturę, realizujące następujące monoidy: liczby całkowite z zerem i dodawaniem, oraz listy z listą pustą i scalaniem list.

Ćwiczenie 4.

Używając `Quickcheck`, sprawdź, że dla komponentów zdefiniowanych w poprzednim zadaniu zachodzą następujące własności:

- `neutral` jest lewostronnym i prawostronnym elementem neutralnym operacji `oper`,
- operacja `oper` jest łączna.

Ćwiczenie 5.

Zdefiniuj sygnaturę z kontraktami opisującą zbiory liczb całkowitych, z predykatem bycia zbiorem, predykatem sprawdzającym przynależność liczby do zbioru, zbiorem pustym, procedurą tworzącą singleton (zbiór jednoelementowy), procedurą sumy zbiorów oraz procedurą iloczynu zbiorów.

Zaimplementuj tę sygnaturę używając list.

Ćwiczenie 6.

Używając `Quickcheck`, sprawdź następujące własności dla komponentu z poprzedniego zadania:

- zbiór pusty nie zawiera żadnego elementu,
- singleton zawiera tylko element użyty do jego konstrukcji,
- suma zbiorów zawiera wyłącznie elementy występujące w co najmniej jednym ze zbiorów podanych jako argumenty,
- iloczyn zbiorów zawiera wyłącznie elementy występujące w obu zbiorach podanych jako argumenty,
- suma zbiorów jest rozdzielna względem iloczynu zbiorów,
- iloczyn zbiorów jest rozdzielny względem sumy zbiorów.

Grafy

Graf to matematyczna struktura opisująca relacje pomiędzy obiektami. Składa się z dwóch zbiorów: niepustego zbioru wierzchołków V oraz zbioru krawędzi $E \subseteq V \times V$. Grafy są bardzo użyteczną abstrakcją – mogą opisywać m.in. połączenia w sieci (komputerowej, drogowej, znajomości...), możliwe zmiany stanu automatów, przepływ sterowania w programach komputerowych, referencje między obiektami w pamięci, itd.

Chcąc operować na danych w postaci grafu, często istnieje potrzeba odwiedzenia wszystkich jego wierzchołków w określonej, zależnej od krawędzi grafu, kolejności. Na przykład, aby policzyć (liczone w liczbie krawędzi) odległości wierzchołków grafu od wybranego wierzchołka, należy odwiedzać wierzchołki rozpoczynając od tych połączonych z nim pojedynczą krawędzią, kolejno przechodząc do coraz bardziej oddalonych wierzchołków. Taką strategię nazywamy

przeszukiwaniem wszerz. Do innych zadań (np. sortowania topologicznego, szukania spójnych składowych) należy zawsze przechodzić do dowolnego jeszcze nie odwiedzonego wierzchołka połączonego bezpośrednio z bieżąco rozpatrywanym, i wycofywać się tylko wtedy, gdy nie można takiego ruchu wykonać (tj. wszystkie wierzchołki bezpośrednio połączone z bieżąco rozpatrywanym są już odwiedzone). Taka strategia jest nazywana przeszukiwaniem w głąb.

Jak się okazuje, algorytmy realizujące przeszukiwanie wszerz i w głąb różnią się tylko strukturą danych zastosowaną do zapamiętania zbioru wierzchołków do odwiedzenia w przyszłości. Gdy użyjemy kolejki FIFO (*first in, first out*), otrzymujemy przeszukiwanie wszerz, natomiast gdy użyjemy stosu, otrzymujemy przeszukiwanie w głąb.

Zadania domowe

Zadanie 20

Następująca sygnatura opisuje struktury danych będące zbiorami elementów, do których można dodawać nowe elementy oraz usuwać je w kolejności ustalonej przez strukturę danych:

```
(define-signature bag^
  ((contracted
    [bag?       (-> any/c boolean?)]
    [empty-bag  (and/c bag? bag-empty?)]
    [bag-empty? (-> bag? boolean?)]
    [bag-insert (-> bag? any/c (and/c bag? (not/c bag-empty?)))]
    [bag-peek   (-> (and/c bag? (not/c bag-empty?)) any/c)]
    [bag-remove (-> (and/c bag? (not/c bag-empty?)) bag?)])))
```

Zaimplementuj dwa komponenty implementujące tę sygnaturę, `bag-stack@` (stos) oraz `bag-fifo@` (kolejkę). Komponenty te powinny być wyeksportowane przy użyciu formy **`provide`**. Implementacja stosu może wykorzystywać pojedynczą listę. Implementacja kolejki używająca pojedynczej listy będzie nieefektywna, dlatego kolejkę należy zaimplementować za pomocą dwóch list, *wejściowej* i *wyjściowej*. Ogólna idea jest taka, że nowe elementy są dodawane `cons`-em do listy wejściowej, natomiast elementy listy wyjściowej są wyjmowane rozpoczynając od pierwszego. Jeśli lista wyjściowa opróżni się, należy w jej miejsce wstawić odwróconą listę wejściową, natomiast w miejsce listy wejściowej wstawić listę pustą.

Wskazówka: Taką kolejkę wygodnie zaimplementować, pisząc konstruktor kolejki, który gdy otrzyma listę pustą w miejscu listy wyjściowej, zamiast tego tworzy kolejkę z listą wyjściową równą odwróconej liście wejściowej. Ten

konstruktor należy zastosować do zaimplementowania procedur zwracających nowe kolejki (`bag-insert` oraz `bag-remove`).

Na SKOS został umieszczony plik `graph.rkt`, zawierający podaną powyżej sygnaturę `bag^`, sygnatury `graph^` i `graph-search^`, przykładową implementację grafów `simple-graph@` i przeszukiwania grafu `graph-search@`. Ten sam plik będzie dostępny w systemie Web-CAT. Należy wykorzystać ten plik przez użycie `(require "graph.rkt")`, **nie należy wklejać jego treści do swojego kodu**. Na SKOS jest również plik `stack-and-fifo.rkt`, który możesz potraktować jako szablon rozwiązania.

Obie implementacje należy przetestować wykorzystując pakiet `quickcheck`. W kodzie załączonym na SKOS znajduje się jeden test dla obu struktur danych, zadaniem jest dopisać własne. Uwaga: testy dla kolejek i stosów mogą się różnić, ponieważ te struktury, mimo implementowania wspólnego interfejsu, mają inne własności!