Lista zagadnień nr 12

Przed zajęciami

W poprzednim tygodniu usłyszeliśmy o abstrakcji **komponentów** (units) oraz mechanizmie **kontraktów**. Tematem bieżącego tygodnia są natomiast **systemy typów** na przykładzie typowanego Racketa. Przed zajęciami należy zapoznać się z kodem źródłowym z wykładu, przejrzeć **rozdziały 7 i 14** przewodnika po Rackecie **The Racket Guide** (`https://docs.racket-lang.org/guide/`) oraz przewodnik po typowanym Rackecie **The Typed Racket Guide** (`https://docs.racket-lang.org/ts-guide/`).

Kontrakty i komponenty

Ćwiczenie 1.

Napisz funkcję `suffixes`, zwracającą wszystkie sufiksy listy podanej jako parametr. Napisz dla tej funkcji odpowiedni kontrakt parametryczny (tzn. wykorzystujący `new-∀/c`).

Ćwiczenie 2.

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

Ćwiczenie 3.

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

Ćwiczenie 4.

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 5.

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.

Racket z typami

Ćwiczenie 6.

Napisz funkcję `prefixes`, zwracającą wszystkie prefiksy listy podanej jako parametr. Nadaj tej funkcji właściwy typ polimorficzny (tzn. wykorzystujący `All`).

Ćwiczenie 7.

Zdefiniuj w typowanym Rackecie typ drzew *rose trees* – to znaczy takich, których liście nie zawierają elementów, natomiast węzły posiadają jedną wartość oraz listę poddrzew. Podobnie jak typ drzew BST z wykładu, zdefiniowany typ powinien być sparametryzowany typem elementu. Zaimplementuj funkcję zwracającą listę elementów takiego drzewa w kolejności preorder.

Ćwiczenie 8.

Rozszerz język z wykładu o ciągi znaków. W tym celu rozszerz typ literałów `Literal` oraz typ wartości `Value` o typ ciągów znaków `String`. Dodaj też operację binarną `append` konkatenacji ciągów znaków do typu `BinopSym`. Rozbuduj odpowiednio interpreter oraz algorytm sprawdzania typów.

Ćwiczenie 9.*

Rozszerz język z wykładu o pary. W tym celu dodaj do niego konstruktor pary `pair-expr`, selektory `first-expr` i `second-expr`. Rozszerz typ wartości `Value` o pary `pair-val`; typem elementów pary mają być wartości. Rozszerz typ typów języka z wykładu `EType` o typ par `pair-type`. Rozbuduj interpreter oraz algorytm sprawdzania typów o obsługę par.

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.

Zadanie domowe (na pracownię)

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). 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 usuwamy elementy z końca listy wyjściowej. 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`).

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!

Kod załączony na SKOS zawiera prostą implementację algorytmu przeszukiwania grafów sparametryzowaną strukturą danych. Należy opracować kilka prostych przykładów grafów i uruchomić na nich obie instancje algorytmu.