Programowanie Funkcyjne 2020

Lista zadań nr 7 dla grup mabi, mbu, ppo i efes

Na zajęcia 1 i 2 grudnia 2020

Zadanie 1 (6p). Zapoznaj się z modułami `Set` oraz `Map` z biblioteki standardowej. Następnie zaimplementuj moduł `Perm` implementujący skończone permutacje (bijekcje σ, takie że jest tylko skończenie wiele elementów x takich, że $\sigma(x) \neq x$). Powinieneś utworzyć plik `Perm.ml`, dla którego plik `Perm.mli` ma następującą zawartość.

```
module OrderedType = sig
  type t
  val compare : t -> t -> int
end

module type S = sig
  type key
  type t
  (** permutacja jako funkcja *)
  val apply   : t -> key -> key
  (** permutacja identycznościowa *)
  val id      : t
  (** permutacja odwrotna *)
  val invert  : t -> t
  (** permutacja która tylko zamienia dwa elementy miejscami *)
  val swap    : key -> key -> t
  (** złożenie permutacji (jako złożenie funkcji) *)
  val compose : t -> t -> t
  (** porównywanie permutacji *)
  val compare : t -> t -> int
end

module Make(Key : OrderedType) : S with type key = Key.t
```

Wygodnie będzie reprezentować permutacje jako pary map skończonych (moduł `Map`), reprezentujące odpowiednio funkcję i funkcje do niej odwrotną. Jeśli danemu kluczowi nie jest przypisana żadna wartość w mapie, to funkcja na tym kluczu działa jako identyczność. Warto utrzymywać niezmiennik, że kluczowi k przypisujemy wartość v tylko wtedy gdy $k \neq v$. Ułatwi to implementację funkcji `compare` (zobacz specyfikację funkcji `compare` z modułu `Map`). Do implementacji funkcji `compose` przyda się funkcja `merge` z modułu `Map`. Dlaczego nasza reprezentacja permutacji powinna być typem abstrakcyjnym?

Zadanie 2 (5p). Napisz funkcję `is_generated`, która sprawdza czy dana permutacja należy do podgrupy permutacji generowanej przez zadaną listę generatorów. Innymi słowy, funkcja dla danej permutacji p oraz listy permutacji g rozstrzyga, czy permutację p można utworzyć za pomocą identyczności, odwracania i składania permutacji znajdujących się na liście g. W tym celu użyj techniki *nasycania zbioru*, która polega na konstruowaniu kolejnych zbiorów permutacji z ciągu opisanego rekurencyjnie

$$\begin{aligned} X_0 &= \{\sigma \mid \sigma \in g\} \cup \{\mathsf{id}\} \\ X_{n+1} &= X_n \cup \{\sigma^{-1} \mid \sigma \in X_n\} \cup \{\sigma_1 \circ \sigma_2 \mid \sigma_1, \sigma_2 \in X_n\} \end{aligned}$$

tak długo aż znajdziemy p w jednym ze zbiorów lub podany ciąg się *nasyci*, tzn. aż znajdziemy takie n, że $X_n = X_{n+1}$. Do obliczania kolejnych elementów tego ciągu przyda się funkcja `fold` z modułu `Set`.

Dodatkowym problemem jaki pojawia się w tym zadaniu jest to, że nasze rozwiązanie powinno działać dla permutacji elementów dowolnego typu. Możesz wybrać jeden z trzech wariantów rozwiązania tego problemu.

- Rozwiąż to zadanie dla ustalonego typu permutacji, np. permutacji liczb typu `int`. W tym celu utwórz moduł

  ```
  module IntPerm = Perm.Make(Int) (** działa z OCamlem w wersji >= 4.08 *)
  ```

  a następnie zdefiniuj funkcję o typie `IntPerm.t -> IntPerm.t list -> bool`. Ten wariant rozwiązania jest warty 2 punkty.
- Zaprogramuj funktor który dla dowolnej implementacji permutacji utworzy moduł zawierający funkcję `is_generated`. Ten wariant jest warty 4 punkty.
- Zapoznaj się z modułami pierwszej kategorii (*first-class modules*), których opis znajdziesz w rozdziale 8.6 dokumentacji języka OCaml. Następnie zaprogramuj funkcję o następującej sygnaturze.

  ```
  val is_generated :
    (module Perm.S with type t = 'a) -> 'a -> 'a list -> bool
  ```

  W tym celu mogą (ale nie muszą) okazać się przydatne typy lokalnie abstrakcyjne (*locally abstract types*) opisane w rozdziale 8.5. Ten wariant jest warty pełne 5 punktów.

Zadanie 3 (4p). Moduły `Set` oraz `Map` z biblioteki standardowej umożliwiają tworzenie zbiorów i map skończonych dla dowolnego typu uporządkowanego. W bibliotece standardowej znajdziemy moduły dostarczające implementacji typów uporządkowanych dla podstawowych typów wbudowanych (np. moduły `Float`, `String`, `Int64`, nawet `Set`, a od niedawna również `Int`). Niestety dla par, list czy typu `option` brakuje takich implementacji. Napisz funktory, które uzupełnią te braki. Np. dla list powinieneś napisać funktor o sygnaturze

```
module OrderedList (X : OrderedType) : OrderedType with type t = X.t list
```

Zadanie 4 (5p). W OCamlu mamy dwa niezależne mechanizmy, które pozwalają parametryzować typy innymi typami: typy polimorficzne (np. `'a list`) oraz funktory. Te pierwsze są znacznie prostsze w użyciu, ale mają pewne ograniczenia. Na przykład typy nie mogą być parametryzowane konstruktorami typów, więc poniższa definicja nie jest poprawna.

```
type 't pair = int 't * string 't
let (x : list pair) = ([42], [])
```

System modułów nie ma tego ograniczenia, ale za to jest dużo cięższy składniowo. Powyższy przykład można przetłumaczyć na system modułów w następujący sposób.

```
module Pair(X : sig type 'a t end) = struct
  type t = int X.t * string X.t
end
module ListPair = Pair(List)
let (x : ListPair.t) = ([42], [])
```

Celem tego zadania będzie napisanie heterogenicznej mapy skończonej w której zarówno typ opisujący klucze jak i wartości ma jeden parametr typowy, który musi się zgadzać w obrębie jednej pary klucz-wartość, ale nie koniecznie w obrębie całej mapy. Zaimplementuj moduł `Map1` o następującej sygnaturze.

```
type ('a, _) order1 =
| Lt : ('a, 'b) order1
| Eq : ('a, 'a) order1
| Gt : ('a, 'b) order1

module type Type1 = sig
```

```
  type 'a t
end

module type OrderedType1 = sig
  include Type1
  val compare : 'a t -> 'b t -> ('a, 'b) order1
end

module type S = sig
  type 'a key
  type 'a value
  type t
  val empty  : t
  val add    : 'a key -> 'a value -> t -> t
  val remove : 'a key -> t -> t
  val find   : 'a key -> t -> 'a value
end

module Make(Key : OrderedType1)(Value : Type1) :
  S with type 'a key = 'a Key.t and type 'a value = 'a Value.t
```

Definicja typu `order1` jest uogólnionym algebraicznym typem danych (*generalized algebraic datatype*, *GADT*) i możesz o nim myśleć jak o typie

```
type ('a, 'b) order1 = Lt | Eq | Gt
```

z dodatkowym ograniczeniem mówiącym o tym, że konstruktora `Eq` można używać tylko wtedy gdy parametry `'a` oraz `'b` są równe (więcej o GADT możesz poczytać w rozdziale 8.11 dokumentacji).

Naturalne przykłady wykorzystujące takie mapy również wymagają użycia GADT. Poniżej znajduje się definicja mapy, która reprezentacjom typów przypisuje przykładowe wartości.

```
type _ typ =
| Int  : int typ
| List : 'a typ -> 'a list typ

module Type = struct
  type 'a t = 'a typ
  let rec compare : type t1 t2. t1 typ -> t2 typ -> (t1, t2) order1 =
    fun tp1 tp2 ->
    match tp1, tp2 with
    | Int,    Int    -> Eq
    | Int,    List _ -> Lt
    | List _, Int    -> Gt
    | List a, List b ->
      begin match compare a b with
      | Eq -> Eq
      | Lt -> Lt
      | Gt -> Gt
      end
end

module TMap = Map1.Make(Type)(struct type 'a t = 'a end)
let m =
  TMap.empty
  |> TMap.add Int        42
  |> TMap.add (List Int) [1;2;3]
```

Do implementacji użyj modułu `Map` z biblioteki standardowej. Będziesz potrzebował opakować klucze i wartości w typy monomorficzne przy użyciu typów egzystencjalnych. Całe szczęście z użyciem GADT jest to bardzo proste.

```
type ex_key =
| Key : 'a key -> ex_key

type key_value_pair =
| KeyVal : 'a key * 'a value -> key_value_pair
```

Mapy heterogeniczne można teraz wyrazić jako zwykłe mapy.

```
module ExKey = struct
  type t = ex_key
  let compare = (* TODO *)
end
module ExMap = Map.Make(ExKey)
type t = key_value_pair ExMap.t
```

Na koniec jeszcze drobna wskazówka. Implementacja funkcji `find` może przysporzyć trochę kłopotów z przekonaniem OCamla, że typy się zgadzają. Będziesz potrzebował dwóch zabiegów, by to osiągnąć.

- Należy sprawdzić za pomocą funkcji `Key.compare` czy poszukiwany klucz jest na pewno równy kluczowi w znalezionej parze klucz-wartość.
- Definiując funkcję `find` trzeba podać odpowiednią anotację typową używając typów lokalnie abstrakcyjnych:

  ```
  let find (type a) (k : a key) m : a value = (* TODO *)
  ```

Uwaga: To jest jedno z zadań, które wyglądają na trudniejsze niż są w rzeczywistości. Ważne jest żeby trzymać się powyższego szablonu, który podpowiada jak poradzić sobie z częścią potencjalnych problemów z typami i, zwłaszcza w razie problemów, doczytać wskazane fragmenty dokumentacji.

Zadanie 5 (dodatkowe, 3p). Na listach 4 i 5 implementowaliśmy prostego asystenta dowodzenia. Pozwalał on na dowodzenie twierdzeń w teoriach które mają skończenie wiele aksjomatów: aksjomaty stawały się założeniami twierdzenia. Niestety wiele interesujących teorii ma nieskończenie wiele aksjomatów, choć dających się opisać w skończony sposób. Na przykład:

- arytmetyka Peano ma nieskończenie wiele zasad indukcji — po jednej dla każdej formuły z wyróżnioną zmienną wolną;
- klasyczny rachunek zdań zanurzony w logice intuicjonistycznej ma nieskończenie wiele aksjomatów odpowiadających prawu podwójnej negacji — po jednym dla każdej formuły.

Zaproponuj rozwiązanie tego problemu przy użyciu funktorów: implementacja logiki może być parametryzowana modułem opisującym teorię. Teoria może zawierać typ reprezentujący aksjomaty, zaś sama implementacja logiki powinna udostępniać jeszcze jedną funkcję

```
val axiom : Theory.axiom -> theorem
```

odpowiadającą regule wnioskowania

$$\frac{\varphi \text{ jest aksjomatem}}{\vdash \varphi}\,(\textsc{Axiom}).$$

Uwaga: Nie musisz mieć rozwiązanej list 4 czy 5 by rozwiązać to zadanie. Możesz rozbudować szablon rozwiązania znajdujący się w SKOSie tylko o kawałki związane z tym zadaniem, tak by program się kompilował. Jednak jeśli wybierzesz tę drogę, to powinieneś umieć uzasadnić, że dokonane modyfikacje nie uniemożliwiają zaimplementowania brakujących kawałków kodu.