Programowanie Funkcyjne 2024

Lista zadań nr 7

Na zajęcia 3 i 5 grudnia 2024

Zadanie 1 (2p). Obliczenia używające liczb pseudolosowych możemy opisać za pomocą dowolnej monady wzbogaconej o operację `random`. Załóżmy, że mamy moduł o poniższej sygnaturze.

```
module type RandomMonad = sig
  type 'a t
  val return : 'a -> 'a t
  val bind   : 'a t -> ('a -> 'b t) -> 'b t
  val random : int t
end
```

Korzystając z takiej monady, napisz funkcję `shuffle`, która dla podanej listy zwraca jej losową permutację. Jeśli nie dysponujesz implementacją wspomnianego modułu, to napisz funktor o poniższej sygnaturze.

```
module Shuffle(R : RandomMonand) : sig
  val shuffle : 'a list -> 'a list R.t
end
```

Zadanie 2 (2p). Aby uruchomić funkcję z poprzedniego zadania, potrzebujemy implementacji monady o sygnaturze `RandomMonad`. Można ją zaimplementować jako monadę stanu, gdzie tym niejawnie przekazywanym stanem jest liczba całkowita reprezentująca aktualną wartość zarodka losowego.

```
module RS : sig include RandomMonad ... end =
struct
  type 'a t = int -> 'a * int
  ...
end.
```

Obliczenie typu `'a RS.t` jest funkcją, która wartość zarodka losowego (przed wykonaniem obliczenia) przekształca w wynik oraz nową wartość zarodka (po wykonaniu obliczenia). Uzupełnij definicję modułu `RS` o brakujące operacje `return`, `bind` oraz `random`, a następnie użyj jej aby przetestować funkcję z poprzedniego zadania. Będziesz potrzebować funkcji `run : int -> 'a RS.t -> 'a` wykonującej obliczenie losowe z podanym początkowym zarodkiem, więc dodaj ją również do modułu `RS`.

Przykładowa funkcja generująca wartości pseudolosowe jest dana przez kolejne wartości ciągu a_i, gdzie a_0 jest początkowym zarodkiem losowym, a kolejne wyrazy zdefiniowane są następująco:

$$\begin{aligned} b_i &= 16807 \cdot (a_i \bmod 127773) - 2836 \cdot (a_i \div 127773) \\ a_{i+1} &= \begin{cases} b_i, & \text{gdy } b_i > 0; \\ b_i + 2147483647, & \text{w p.p.} \end{cases} \end{aligned}$$

Zadanie 3 (1p). Zaimplementuj funkcje `return` oraz `bind` dla monady identycznościowej, tj. takiej w której typ obliczeń jest typem wyniku:

```
type 'a t = 'a.
```

Powtórz polecenie dla monady odroczonych obliczeń:

```
type 'a t = unit -> 'a.
```

Zadanie 4 (3p). Pokaż, że monady `Err`, `BT`, oraz `St` z wykładu mogą być zaimplementowane jako szczególny przypadek monady kontynuacyjnej. Tzn. dostarcz implementacje modułów o poniższych sygnaturach:

```
module type Monad = sig
  type 'a t
  val return : 'a -> 'a t
  val bind   : 'a t -> ('a -> 'b t) -> 'b t
end
module Err : sig
  include Monad
  val fail  : 'a t
  val catch : 'a t -> (unit -> 'a t) -> 'a t
  val run   : 'a t -> 'a option
end
module BT : sig
  include Monad
  val fail : 'a t
  val flip : bool t
  val run  : 'a t -> 'a Seq.t
end
module St(State : sig type t end) : sig
  include Monad
  val get : State.t t
  val set : State.t -> unit t
  val run : State.t -> 'a t -> 'a
end
```

w taki sposób, by w każdej monadzie typ `'a t`, był typem funkcji, która oczekuje na kontynuację. Ta ostatnia monada jest zdefiniowana jako funktor, którego parametrem jest typ ukrytego stanu.[1] Może się okazać, że będzie potrzebny odpowiedni polimorfizm ze względu na typ odpowiedzi. Aby rozwiązać ten problem, możesz użyć polimorficznych pól rekordów i zdefiniować typ `'a t` w następujący sposób:

```
type 'a t = { run : 'r. ('a -> 'r ans) -> 'r ans }
```

dla odpowiedniego typu `'r ans` (ten typ będzie inny dla każdej z monad).

Zadanie 5 (3p). Wyrażenia regularne można opisać następującym typem danych.

```
type 'a regexp =
  | Eps
  | Lit  of ('a -> bool)
  | Or   of 'a regexp * 'a regexp
  | Cat  of 'a regexp * 'a regexp
  | Star of 'a regexp
```

Dla wygody wprowadźmy jeszcze dwa operatory binarne, do zapisu konstruktorów `Or` oraz `Cat`.

```
let ( +% ) r1 r2 = Or(r1, r2)
let ( *% ) r1 r2 = Cat(r1, r2)
```

W naszej definicji wyrażeń regularnych parametr typowy `'a` oznacza alfabet nad którym pracujemy (zwykle będzie to typ `char`). Znaczenia konstruktorów są następujące.

`Eps` dopasowuje się tylko do słowa pustego.

`Lit p` dopasowuje się tylko do jednoliterowych słów, których jedyna litera spełnia predykat `p`.

[1] Alternatywnie, typ `t` w monadzie `St` może mieć dwa parametry — typ wyniku obliczenia i typ ukrytego stanu, ale wtedy nie mógłby rozszerzać sygnatury `Monad` w której typ `t` ma tylko jeden parametr.

`r1 +% r2` dopasowuje się tylko do słów, które są opisane przynajmniej jednym z wyrażeń `r1` oraz `r2`.

`r1 *% r2` dopasowuje się do słów które można utworzyć poprzez konkatenację słowa opisanego przez `r1` ze słowem opisanym przez `r2`.

`Star r` dopasowuje się do słów, które można rozbić na ciąg słów (być może pusty), z których każde pasuje do `r`.

Na przykład słowa, w których bezpośrednio po każdej literce `'b'` występuje `'a'` można opisać następującym wyrażeniem regularnym.

```
Star (Star (Lit ((<>) 'b')) +% (Lit ((=) 'b') *% Lit ((=) 'a')))
```

Korzystają z monady `BT` (tej z wykładu albo tej z zadania 4) napisz funkcję, która próbuje dopasować wyrażenie regularne do prefiksu danego słowa.

```
match_regexp : 'a regexp -> 'a list -> 'a list option BT.t
```

Aby uniknąć zapętlenia dla gwiazdki na pustym słowie, obliczenie powinno zwracać `None`, gdy dopasowano się do pustego słowa oraz `Some xs`, gdy dopasowano się do niepustego słowa, a pozostały sufiks to `xs`.

Zadanie 6 (4p). Chcemy zaimplementować monadę, która implementuje obliczenia ze stanem i nawracaniem, czyli chcemy połączyć interfejs monad `St` oraz `BT`.

```
module SBT(State : sig type t end) : sig
  type 'a t
  val return : 'a -> 'a t
  val bind   : 'a t -> ('a -> 'b t) -> 'b t
  val fail   : 'a t
  val flip   : bool t
  val get    : State.t t
  val put    : State.t -> unit t
  val run    : State.t -> 'a t -> 'a Seq.t
end
```

Okazuje się, że te dwa efekty można połączyć ze sobą na dwa sposoby — w zależności od tego, czy wartość mutowalnej komórki jest przywracana wraz z nawrotami, czy nie. W tym zadaniu wybierzemy ten łatwiejszy wariant, tzn. taki, gdzie typ `'a t` ma poniższą definicję:

```
type 'a t = State.t -> ('a * State.t) Seq.t.
```

Zauważ, że jest to taki wariant monady stanu, gdzie zwracamy całą sekwencję możliwych wyników (i nowych stanów). Zaimplementuj wszystkie funkcje wymienione w sygnaturze tej monady, a następnie zastanów się, który z dwóch wariantów połączenia stanu z nawracaniem zaimplementowałeś.

Jeśli rozwiązałeś zadanie 5, to teraz możesz zademonstrować użyteczność monady `SBT`, ukrywając przetwarzany ciąg znaków jako stan obliczeń.

Zadanie 7 (3p). W tym zadaniu chcemy połączyć stan z nawracaniem na drugi z możliwych sposobów, czyli chcemy dostarczyć alternatywną implementację modułu `SBT` z poprzedniego zadania. Do reprezentacji obliczeń użyj następującego typu:

```
type 'a t = State.t -> 'a sbt_list * State.t
and 'a sbt_list =
  | Nil
  | Cons of 'a * 'a t
```

i zaimplementuj wszystkie funkcje z modułu `SBT`. Czy potrafisz zademonstrować użyteczność tej monady?

Wskazówka: Znacznie łatwiej jest rozwiązać to zadanie, gdy wiesz co robisz. Zanim zaczniesz bić się z systemem typów, zastanów się który ze wspomnianych dwóch wariantów powinieneś zaimplementować i jak stan jest przekazywany pomiędzy poszczególnymi kawałkami obliczenia.

Zadanie 8 (2p, łatwe!). Dotychczas patrzyliśmy na monady jak na sposób opisu obliczeń z efektami. Alternatywnie, na monady można patrzeć jak na drzewa, które trzymają dane w liściach. Takie drzewa są wszechobecne zarówno w informatyce jak i logice. Na przykład poniższy typ, reprezentujący termy logiki pierwszego rzędu może być postrzegany jako pewien typ drzew, gdzie danymi są zmienne.

```
type symbol = string
type 'v term =
  | Var of 'v
  | Sym of symbol * 'v term list
```

Odkryj w nim strukturę monady i zaimplementuj funkcje `return` oraz `bind`. Następnie zastanów się jakie znane nam operacje na termach kryją się za tymi funkcjami?