Programowanie Funkcyjne 2018

Lista zadań nr 4 dla grupy TWI: algebraiczne typy danych

Na zajęcia 8 listopada 2018

Algebraiczne typy danych bardzo przydają się do przedstawiania składni abstrakcyjnej różnych języków. Rozważmy na przykład zbiór formuł $\mathcal{F}(V)$ rachunku zdań znany z kursu logiki:

```
type 'var prop = Var of 'var
               | Not of 'var prop
               | And of 'var prop * 'var prop
               | Or  of 'var prop * 'var prop
               | Imp of 'var prop * 'var prop
```

$$
\begin{aligned}
v \in V &\implies v \in \mathcal{F}(V) \\
\phi \in \mathcal{F}(V) &\implies \neg\phi \in \mathcal{F}(V) \\
\phi, \psi \in \mathcal{F}(V) &\implies \phi \wedge \psi \in \mathcal{F}(V) \\
\phi, \psi \in \mathcal{F}(V) &\implies \phi \vee \psi \in \mathcal{F}(V) \\
\phi, \psi \in \mathcal{F}(V) &\implies \phi \Rightarrow \psi \in \mathcal{F}(V)
\end{aligned}
$$

Po prawej stronie znajduje się definicja z *Whitebooka*, po lewej — jej Ocamlowa wersja. Zbiór zmiennych V nazwaliśmy w Ocamlu `'var`, zbiór formuł $\mathcal{F}(V)$ — `'var prop` (typy z argumentami zapisujemy w Ocamlu postfiksowo). Ponieważ Ocaml jest językiem z aplikatywną strategią ewaluacji, to zbiór wartości typu `'var prop` jest izomorficzny ze zbiorem $\mathcal{F}(V)$ zdefiniowanym w *Whitebooku*, własności typu `'var prop` możemy dowodzić przez indukcję strukturalną, a funkcje działające na wartościach tego typu możemy definiować za pomocą rekursji strukturalnej (działa bowiem znane z *Whitebooka* twierdzenie o definiowaniu przez indukcję strukturalną). Ta odpowiedniość nie jest słuszna w przypadku języków *non-strict*, takich jak np. Haskell. W definicjach poniżej będziemy używać tylko notacji Ocamlowej pamiętając, że są to w istocie zwykłe znane z matematyki definicje indukcyjne.

Dla dowolnego zbioru zmiennych `'var` definiujemy za *Whitebookiem* zbiór *literałów* następująco:

```
type 'var lit = Pos of 'var | Neg of 'var
```

Formuły w *negacyjnej postaci normalnej* możemy zaś zdefiniować następująco:

```
type 'var nnf = LitNNF of 'var lit
              | AndNNF of 'var nnf * 'var nnf
              | OrNNF  of 'var nnf * 'var nnf
```

Klauzule to zbiory literałów. Zbiory możemy symulować w Ocamlu za pomocą list (ignorując kolejność i nie pozwalając na powtórzenia). Mamy wtedy:

```
type 'var clause = 'var lit list
```

Formuły w *koniunkcyjnej postaci normalnej* to zbiory klauzul:

```
type 'var cnf = 'var clause list
```

Zadanie 1 (2 pkt). Jako składnię konkretną formuł rachunku zdań wybieramy odwrotną notację polską (notację postfiksową), w której spójniki logiczne $\neg$, $\wedge$, $\vee$ i $\Rightarrow$ oznaczamy znakami `~`, `*`, `+` i `>`, a zbiór zmiennych, to zbiór małych liter `a`–`z`, np. `pq~r*+` oznacza formułę $p \vee (\neg q \wedge r)$. Zaprogramuj funkcje

```
parse_rpn : string -> char prop
unparse_rpn : char prop -> string
```

Funkcja `parse_rpn` nie jest określona na całym zbiorze napisów, tylko na zbiorze napisów poprawnych składniowo. Jeśli jej argument nie należy do tego zbioru, to powinna zgłosić odpowiedni wyjątek.

Zaletą notacji polskiej jest to, że powyższe funkcje z, odpowiednio, dziedziną i zbiorem wartości ograniczonymi do zbioru napisów poprawnych składniowo są wzajemnie odwrotnymi bijekcjami.

Zadanie 2 (2 pkt). Jako składnię konkretną formuł rachunku zdań wybieramy notację, w której najsilniej wiąże prefiksowy operator negacji `~`. Operatory infiksowe koniunkcji `*` i alternatywy `+` wiążą w lewo, przy czym alternatywy słabiej niż koniunkcji. Najsłabiej oraz w prawo wiąże operator implikacji `>`. Zbiór zmiennych, to zbiór małych liter `a`–`z`. W zapisie formuł można używać nawiasów. Dla przykładu napis `(p+~q)*r` oznacza formułę $(p \vee \neg q) \wedge r$. Zaprogramuj funkcje

```
parse_prop : string -> char prop
unparse_prop : char prop -> string
```

Funkcja `parse_prop` nie jest określona na całym zbiorze napisów, tylko na zbiorze napisów poprawnych składniowo. Jeśli jej argument nie należy do tego zbioru, to powinna zgłosić odpowiedni wyjątek. Funkcja `unparse_prop` powinna używać nawiasów tylko tam, gdzie to konieczne, biorąc pod uwagę siłę i kierunek łączności operatorów.

Zauważ, że `parse_prop` nie jest różnowartościowa na zbiorze napisów poprawnych składniowo (jest tylko surjekcją) i że `unparse_prop` nie jest surjekcją na zbiór napisów poprawnych składniowo (jest tylko różnowartościowa). Złożenie `unparse_prop` z `parse_prop` jest identycznością na zbiorze formuł `char prop`, ale funkcja

```
let f str = unparse_prop (parse_prop str)
```

nie jest identycznością na zbiorze napisów poprawnych składniowo. Ma natomiast całkiem ciekawą interpretację. Jaką? Jak powinna nazywać się ta funkcja?

Zadanie 3 (2 pkt). Zaprogramuj funkcję

```
nnf_of_prop : 'var prop -> 'var nnf
```

sprowadzającą podaną formułę do negacyjnej postaci normalnej. W zeszłorocznym *Whitebooku* robili to inaczej niż w tegorocznym. Ciekawe czemu? (Kopie *Whitebooków* są dostępne na stronie zajęć).

Zadanie 4 (2 pkt). Zaprogramuj funkcję

```
cnf_of_prop : 'var prop -> 'var cnf
```

sprowadzającą podaną formułę do koniunkcyjnej postaci normalnej.

Zadanie 5 (1 pkt). Zaprogramuj funkcję

```
var_of_prop : 'var prop -> 'var list
```

ujawniającą listę wszystkich zmiennych występujących w podanej formule, w dowolnej kolejności, ale bez powtórzeń (do wykrycia powtórzeń użyj operatora porównania fizycznego).

Zadanie 6 (1 pkt). Zaprogramuj funkcję

```
subst_prop : ('v -> 'w prop) -> 'v prop -> 'w prop
```

która za zmienne (typu `'v`) podstawia w formule (typu `'v prop`) formuły (typu `'w prop`) dając w wyniku formułę (typu `'w prop`). Na przykład wyrażenie

```
subst_prop (fun v -> Var (int_of_char v)) phi
```

gdzie

```
phi = And (Var 'p', Var 'q') : char prop
```

ma wartość

```
And (Var 112, Var 113) : int prop
```

zaś wyrażenie

```
subst_prop (function 'p' -> Or (Var 'q', Var 's') | v -> Var v) phi
```

ma wartość

```
And (Or (Var 'q', Var 's'), Var 'q') : char prop
```

Zadanie 7 (1 pkt). Zaprogramuj funkcję

```
valuation : ('v -> bool) -> 'v prop -> bool
```

wyznaczającą wartość logiczną podanej formuły dla podanego wartościowania typu `'v -> bool`. Zauważ, że `valuation`(σ) to w notacji z *Whitebooka* $\hat{\sigma}$. Zauważ podobieństwo funkcji `subst_prop` i `valuation` — obie mają typ postaci $(\texttt{'v} \to \sigma) \to (\texttt{'v prop} \to \sigma)$, tj. przedłużają homomorficznie podaną funkcję określoną na zbiorze zmiennych (będącym generatorem zbioru termów) do funkcji określonej na zbiorze wszystkich termów o wartościach w pewnej algebrze σ o tej samej sygnaturze.

Zadanie 8 (1 pkt). W tym zadaniu użyjemy list asocjacji do zdefiniowania wartościowań zmiennych. Niech

```
type varval = (char * bool) list
exception Unvalued of char
```

Zaprogramuj funkcję

```
getval : varval -> char -> bool
```

ujawniającą wartościowanie podanej zmiennej. Funkcja ta zgłasza wyjątek `Unvalued` jeśli zmiennej nie przypisano wartości. Dla przykładu wyrażenie

```
getval [('p',true), ('q',false), ('r',false)] 'q'
```

ma wartość `false`. Zauważ, że jeśli `v : varval`, to `getval v` jest wartościowaniem odpowiedniego typu, żeby skorzystać z funkcji `valuation`. Zatem `getval` to sposób na zwięzłe definiowanie wartościowań (inne rozwiązanie polegałoby na skorzystaniu z bibliotecznego modułu `Map`).

Zadanie 9 (1 pkt). Zaprogramuj funkcję

```
nextval : varval -> varval option
```

wyznaczającą dla podanego wartościowania kolejne (i zwracającą `None` dla ostatniego wartościowania). Kolejność wartościowań jest następująca: pierwsze wartościowanie przypisuje wszystkim zmiennym wartość `false`, ostatnie zaś przypisuje wszystkim zmiennym wartość `true`. Jeśli wartości przypisane zmiennym ustawimy w takiej kolejności, w jakiej występują na liście `varval` i potraktujemy `false` jako cyfrę binarną 0, a `true` jako 1, to wyznaczenie kolejnego wartościowania sprowadza się do wyznaczenia następnika liczby w zapisie binarnym, w którym cyfry są zapisane w kolejności od najmniej do najbardziej znaczącej. Na przykład wyrażenie

```
nextval [('p',true), ('q',true), ('r',false)]
```

ma wartość

```
Some [('p',false), ('q',false), ('r',true)]
```

Zadanie 10 (1 pkt). Użyj funkcji z poprzednich zadań do zdefiniowania funkcji

```
sat_prop : char prop -> bool
```

która odpowiada na pytanie, czy podana formuła jest spełnialna.

W kolejnych zadaniach wykorzystamy algebraiczne typy danych do reprezentowania drzew. Niech zatem

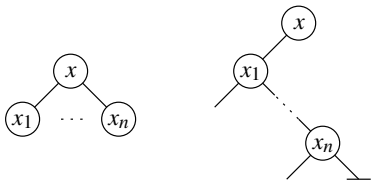
```
type 'a btree = Leaf | Node of 'a btree * 'a * 'a btree
type 'a mtree = MTree of 'a * 'a forest
 and 'a forest = 'a mtree list
```

Zadanie 11 (1 pkt). Zaprogramuj funkcje

```
bprefix : 'a btree -> 'a list
mprefix : 'a mtree -> 'a list
```

ujawniające listy etykiet drzew w porządku prefiksowym. Ze wzgledu na nieliniową rekursję o ogonowość nie ma tu co walczyć (głębokość rekursji jest proporcjonalna do głębokości drzewa, więc dla drzew bliskich zbalansowaniu niewielka), ale postaraj się nie generować nieużytków (do tego celu potrzebne będą pewnie akumulatory).

Zadanie 12 (2 pkt). Drzewa o zmiennej liczbie potomków można zastąpić drzewami binarnymi zgodnie ze schematem przedstawionym na rysunku:



W tej transformacji prawy sąsiad staje się prawym potomkiem, lewy sąsiad staje się ojcem, jedynie skrajnie lewy syn pozostaje na swoim miejscu. Wierzchołek, który nie miał prawego sąsiada staje się wierzchołkiem, który nie ma prawego syna. Transformacja odpowiada obróceniu rysunku drzewa o 45° zgodnie z ruchem wskazówek zegara. Zaprogramuj funkcje

```
btree_of_mtree : 'a mtree -> 'a btree
mtree_of_btree : 'a btree -> 'a mtree
```

Ile miejsca oszczędzamy na tej transformacji? Jak wyznaczyć listę etykiet w porządku prefiksowym drzewa o zmiennej liczbie potomków reprezentowanego za pomocą drzewa binarnego?

Zadanie 13 (2 pkt). Drzewo jest *wyważone*, jeśli długości najdłuższej i najkrótszej ścieżki od korzenia do liścia różnią się co najwyżej o jeden. Napisz funkcję

```
const_tree : 'a -> int -> 'a btree
```

która buduje wyważone drzewo o podanej liczbie wierzchołków, w którym wszystkie etykiety mają tę samą, podaną wartość. Wykorzystaj maksymalnie współdzielenie — powinieneś zaalokować pamięć dla co najwyżej $2h$ węzłów `Node`, gdzie h jest wysokością drzewa (równą $\lfloor \log n \rfloor + 1$, gdzie n jest liczbą wierzchołków). Nie powinieneś generować nieużytków. Rekursja, oczywiście, nie musi być ogonowa (jeśli jej głębokość jest proporcjonalna do wysokości drzewa).

Zadanie 14 (1 pkt). Zaprogramuj funkcję

```
palindorme : 'a list -> bool
```

odpowiadającą na pytanie, czy podana lista jest palindromem (względem relacji porównania fizycznego). Nie wolno generować *żadnych* nieużytków, a głębokość rekursji powinna być ograniczona do połowy długości listy. Do sprawdzenia, czy doszliśmy do połowy listy skorzystaj z pomysłu wykorzystanego do zdefiniowania funkcji `split` z jednej z poprzednich list. W kolejnych wywołaniach rekurencyjnych zapamiętuj kolejne elementy listy. Podczas powrotu z rekursji porównuj je z kolejnymi elementami z drugiej połowy listy.

Niniejszy tekst jest kompilacją listy zadań przygotowanej przez Filipa Sieczkowskiego.