

Programowanie Funkcyjne 2025

Lista zadań nr 9

Na zajęcia 4 i 12 grudnia 2025

Zadanie 1 (3p). Rozważmy następujący typ danych opisujący pewien podzbiór typów skończonych.

```
type _ fin_type =
| Unit : unit fin_type
| Bool : bool fin_type
| Pair : 'a fin_type * 'b fin_type -> ('a * 'b) fin_type
```

Zaimplementuj funkcję `all_values`: `'a fin_type -> 'a Seq.t`, która dla podanego typu zwraca wszystkie jego elementy. Na przykład, wywołanie `all_values (Pair(Unit, Bool))` powinno zwrócić sekwencję dwóch par: `(((), true)` oraz `(((), false)`).

Zadanie 2 (1p). Rozszerz typ z poprzedniego zadania o konstruktory opisujące typ pusty i typ `Either.t`. Typ pusty nie jest zdefiniowany w bibliotece standardowej, więc powinieneś go zdefiniować samemu.

```
type empty = |
```

Następnie uzupełnij definicję funkcji `all_values` o obsługę nowych przypadków.

Zadanie 3 (1p). Rozbuduj typ z zadania 1 o konstruktor opisujący funkcje, a następnie uzupełnij definicję funkcji `all_values`. Przyjmij, że typ `fin_type` potrafi opisywać tylko funkcje totalne, tzn. takie, że dla poprawnych danych zawsze się zakończą i zwrócią poprawny wynik. Dodatkowo przyjmij, że dwie funkcje są równe, gdy dla wszystkich poprawnych argumentów dają te same wyniki. Wtedy typ `a -> b` będzie mieć dokładnie b^a elementów, gdzie a i b to odpowiednio liczba elementów typów a i b .

Wskazówka: rozwiązanie prawie na pewno wymaga zdefiniowania drugiej funkcji wzajemnie rekurencyjnej z `all_values`, której typ będzie mieć negatywne wystąpienia zmiennej typowej będącej parametrem `fin_type`. W moim rozwiążaniu jest to funkcja `equal_at`: `'a fin_type -> 'a -> 'a -> bool`.

Zadanie 4 (3p). Kwantyfikowane formuły Boole'owskie (*Quantified Boolean Formulas*, QBF) to formuły logiczne, które oprócz zmiennych zdaniowych i klasycznych spójników logicznych zawierają kwantyfikatory, które wiążą zmienne zdaniowe. Dla uproszczenia przyjmijmy następującą gramatykę QBF, wyrażoną jako algebraiczny typ danych.

```
module QBF = struct
  type var = string
  type formula =
    | Var of var          (* zmienne zdaniowe *)
    | Bot                 (* spójnik fałszu (⊥) *)
    | Not of formula      (* negacja (¬φ) *)
    | And of formula * formula (* koniunkcja (φ ∧ ψ) *)
    | All of var * formula (* kwantyfikacja uniwersalna (∀p.φ) *)
    ...
end
```

Na przykład $(\forall p.p)$ reprezentowane jest jako `All("p", Var "p")` i jest formułą, która jest fałszywa, natomiast formuła $(\forall p. \neg \forall q. \neg (\neg(p \wedge q) \wedge \neg(\neg p \wedge \neg q)))$ jest prawdziwa.

Napisz funkcję `is_true : formula -> bool` sprawdzającą czy podana formuła jest prawdziwa. Możesz założyć, że formuła wejściowa jest zamknięta, tj. nie ma zmiennych wolnych. Jednak gdy zejdziemy pod kwantyfikator, to mogą pojawić się zmienne wolne. W tym celu przyda się pomocnicza funkcja, która ewaluuje formułę w podanym wartościowaniu/środowisku:

```

type env = var -> bool
val eval : env -> formula -> bool

```

Jako, że pracujemy w logice klasycznej, to każda zmienna zdaniowa może być prawdziwa albo fałszywa. Zatem obsłużenie kwantyfikatora jest proste: wystarczy sprawdzić wszystkie dwa przypadki.

Zadanie 5 (4p). Implementacja QBFa z poprzedniego zadania nie jest w pełni zadowalająca, bo nie wymuszamy na użytkowniku, by zawsze podawał zamknięte formuły. Oczywiście zawsze możemy zgłosić błąd gdy napotkamy zmienną wolną, ale problem ten można elegancko rozwiązać przy pomocy *nieregularnych typów danych*, a dokładniej używając techniki reprezentowania wiązania zmiennych za pomocą zagnieżdzonych typów danych (*nested datatypes*). Idea polega na tym, że typ formuł jest sparametryzowany typem używanym do reprezentacji zmiennych wolnych.

```

module NestedQBF = struct
  type 'v formula =
    | Var of 'v
    | Bot
    | Not of 'v formula
    | And of 'v formula * 'v formula
    | All of 'v inc formula
  ...
end

```

Używając typu `empty` z zadania 2 możemy wyrazić zamknięte formuły jako `empty formula`. Gdy napotkamy zmienną reprezentowaną przez typ `empty` możemy użyć funkcji

```
let absurd (x : empty) = match x with | _ -> .
```

by przekonać kompilator, że taka sytuacja jest niemożliwa. Jednak co zrobić z podformułami, które znajdują się pod kwantyfikatorem, skoro mogą one mieć zmienne wolne? Do tego służy następujący typ

```
type 'v inc = Z | S of 'v
```

który każdy element typu `'v` owija w konstruktor `S` oraz dodaje jeden nowy element `Z`, który reprezentuje tą zmienną która jest związana przez kwantyfikator. Na przykład formułę $\forall p.p \wedge (\forall q.p \wedge q)$ wyrazimy jako

```
All(And(Var Z, All(And(Var (S Z), Var Z))))
```

(czy coś Ci to przypomina? Przypomnij sobie o indeksach de Bruijna).

Napisz teraz funkcję `is_true : empty formula -> bool`, która sprawdza czy podana formuła jest prawdziwa. Również przyda się funkcja `eval` ze środowiskiem. Jaki typ powinna mieć funkcja `eval`?

Zadanie 6 (3p). Napisz funkcję konwertującą formuły z zadania 4 do formuł z zadania 5.

```

type 'v env = QBF.var -> 'v
val scope_check : 'v env -> QBF.formula -> 'v NestedQBF.formula

```

Pierwszy argument tej funkcji to środowisko: funkcja mapująca zmienne z jednej reprezentacji na drugą (może zgłosić błąd, gdy zmienna nie jest zdefiniowana). Zauważ, że w tym zadaniu będzie potrzebna rekursja polimorficzna.

Zadanie 7 (4p). Teraz rozszerzymy naszą implementację QBFa o funkcje wyższego rzędu, tak by można było kwantyfikować nie tylko po zmiennych zdaniowych, ale również po relacjach czy też funkcjach Boole'owskich. Na przykład poprawną formułą (i też prawdziwą) jest $\neg\forall f.\neg(f(\perp) \wedge \neg f(\neg\perp))$. Jednak jeśli rozszerzymy gramatykę formuł o aplikację funkcji oraz λ -abstrakcję, to nie wszystkie tak skonstruowane formuły mają sens, na przykład $\forall f.f \wedge f(\perp)$ czy $\perp(\perp)$. Aby odrzucić niepoprawne formuły, dodamy system typów dla formuł logicznych. Do reprezentacji typów użyjemy typów skończonych z zadań 1–3, natomiast cały system typów zakodujemy używając GADT. Oba typy reprezentujące odpowiednio zmienne i formuły będą teraz sparametryzowane dwoma typami, opisującymi odpowiednio środowisko oraz typ danego obiektu. Typy

reprezentujące środowisko będą miały postać listy typów zmiennych, gdzie pustą listę zareprezentujemy przy pomocy typu unit, a consa przy pomocy typu par tp * env. Zatem typy reprezentujące zmienne i formuły wyglądają następująco.

```
type (_, 'a) var =
| Z : ('a * 'b, 'a) var
| S : ('env, 'a) var -> ('b * 'env, 'a) var

type ('env, _) formula =
| Var : ('env, 'a) var -> ('env, 'a) formula
| Bot : ('env, bool) formula
| Not : ('env, bool) formula -> ('env, bool) formula
| And : ('env, bool) formula * ('env, bool) formula -> ('env, bool) formula
| All : 'a fin_type * ('a * 'env, bool) formula -> ('env, bool) formula
| Lam : ('a * 'env, 'b) formula -> ('env, 'a -> 'b) formula
| App : ('env, 'a -> 'b) formula * ('env, 'a) formula -> ('env, 'b) formula
```

Zauważ, że kwantyfikator uniwersalny zawiera teraz informacje o typie wiązanej zmiennej. Doszły nam też dwa nowe konstruktory, jeden reprezentujący λ -abstrakcję i jeden reprezentujący aplikację funkcji.

Napisz funkcję is_true : (unit, bool) formula -> bool, sprawdzającej prawdziwość zamkniętej formuły typu bool. W tym celu zdefiniuj pomocniczą funkcję eval : 'env -> ('env, 'tp) formula -> 'tp. Zwróć uwagę na to, jak wygląda typ środowisk. Przydatna może się okazać poniższa funkcja.

```
let rec lookup : type env a. env -> (env, a) var -> a =
  fun env x ->
  match x with
  | Z   -> fst env
  | S y -> lookup (snd env) y
```

Zadanie 8 (1p). Rozbuduj rozwiązanie poprzedniego zadania o nowe konstrukcje w gramatyce formuł, pozwalające manipulować pozostałymi typami skończonymi, np. projekcje, pary, itp.

Zadanie 9 (ambitne, ?p + chwała i uznanie). Zdefiniuj typ w duchu typu formula z zadania 4 (tj. bez użycia GADT i nieregularnych typów danych) reprezentujący kwantyfikowane formuły Boole'owskie wyższego rzędu, takie jak w zadaniu 7. Następnie podobnie jak w zadaniu 6 zdefiniuj funkcję która taką prostą reprezentacją zamienia na formuły z zadania 7.