Programowanie Funkcyjne 2022

Lista zadań nr 3

Na zajęcia 3, 7 i 8 listopada 2022

Zadanie 1 (4p). 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 type 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 (4p). 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 dwóch wariantów rozwiązania tego problemu.

- Zaprogramuj funktor który dla dowolnej implementacji permutacji utworzy moduł zawierający funkcję `is_generated`.
- Zapoznaj się z modułami pierwszej kategorii (*first-class modules*), których opis znajdziesz w rozdziale 10.5 dokumentacji języka OCaml zamieszczonej w SKOSie. 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 10.4.

Wskazówka: Moduł `Perm` implementuje funkcję `compare`, więc może być parametrem dla funktora `Set.Make` z biblioteki standardowej.

Mini-Projekt: Asystent Dowodzenia, cz. I

Zadanie 3 (1p). W tym oraz następnych zadaniach zajmiemy się fragmentem logiki intuicjonistycznej, w którym formuły składają się tylko ze zmiennych zdaniowych (p,q,r,...), binarnego spójnika implikacji ($\to$) oraz 0-arnego spójnika fałszu ($\bot$).[1] Np. poprawnymi formułami są

$$p \qquad p \to q \qquad (p \to \bot) \to \bot \qquad (p \to q \to r) \to (p \to q) \to p \to r.$$

Wypisując formuły przyjmujemy, że implikacja wiąże w prawo, więc ta ostatnia formuła w istocie oznacza

$$(p \to (q \to r)) \to ((p \to q) \to (p \to r)).$$

Na stronie przedmiotu znajduję się szablon rozwiązania. W plikach `logic.mli` oraz `logic.ml` uzupełnij definicję typu `formula` o zaproponowaną przez siebie reprezentacją formuł w rozważanej logice.

Wskazówka: Zauważ, że formuły to nic innego, jak drzewa binarne, które mają dwa rodzaje liści.

Zadanie 4 (2p). Uzupełnij definicję funkcji `string_of_formula` znajdującą się w pliku `logic.ml`. Funkcja powinna przekształcać formułę w napis czytelny dla człowieka, używający jak najmniejszej liczby nawiasów. Funkcja ta jest prywatna dla modułu `Logic` i służy jedynie zaimplementowaniu funkcji `pp_print_formula`, którą można zarejestrować w interpreterze, jako sposób wyświetlania formuł:

```
utop # #install_printer Logic.pp_print_formula ;;
utop # let f = {zależne od Twojej reprezentacji formuł} ;;
val f : Logic.formula = (p → ⊥) → p → q
```

Uwaga: Jeśli jesteś odważny, możesz rozwiązać trudniejszy wariant tego zadania, polegający na bezpośrednim zaimplementowaniu funkcji `pp_print_formula`. W tym celu zapoznaj się z modułem `Format` z biblioteki standardowej. Za rozwiązanie które ładnie formatuje bardzo duże formuły, można dostać dodatkowe punkty!

Zadanie 5 (3p). Wprowadzimy teraz formalny system dowodzenia do naszej logiki. Będzie to wariant systemu naturalnej dedukcji, znanego Wam z kursu logiki. *Osądem* nazwiemy parę $\Gamma \vdash \varphi$, taką że Γ jest skończonym zbiorem formuł (zwanym *założeniami*), a φ jest formułą (zwaną *tezą* albo *konsekwencją*). Znak $\vdash$ pełni tu tylko rolę przecinka. Dowody są drzewami, zbudowanymi z następujących reguł wnioskowania.

$$\frac{}{\{\varphi\} \vdash \varphi}(\mathrm{Ax}) \qquad \frac{\Gamma \vdash \varphi}{\Gamma \setminus \{\psi\} \vdash \psi \to \varphi}(\to\mathrm{I}) \qquad \frac{\Gamma \vdash \varphi \to \psi \quad \Delta \vdash \varphi}{\Gamma \cup \Delta \vdash \psi}(\to\mathrm{E}) \qquad \frac{\Gamma \vdash \bot}{\Gamma \vdash \varphi}(\bot\mathrm{E})$$

Innymi słowy, dowody to drzewa, których wszystkie wierzchołki są etykietowane osądami i spełniają jeden z warunków:

[1] Rozszerzenie tego rachunku po pełnej logiki intuicjonistycznej z koniunkcją, dysjunkcją i stałą prawdy nie wprowadza istotnych utrudnień — ale dodaje trochę kodu do napisania. Jeśli masz ochotę, rozszerz logikę o dodatkowe konstrukcje!

- wierzchołek nie ma dzieci i jest etykietowany osądem $\{\varphi\} \vdash \varphi$, dla pewnej formuły φ;
- wierzchołek ma tylko jedno dziecko, etykietowane osądem $\Gamma \vdash \varphi$, zaś samo jest etykietowane osądem $\Gamma \setminus \{\psi\} \vdash \psi \to \varphi$;
- wierzchołek ma dwójkę dzieci, których etykiety mają postać odpowiednio $\Gamma \vdash \varphi \to \psi$ oraz $\Delta \vdash \varphi$ (dla pewnych Γ, Δ, φ oraz ψ), zaś sam wierzchołek ma etykietę $\Gamma \cup \Delta \vdash \psi$;
- wierzchołek ma etykietę $\Gamma \vdash \varphi$ oraz tylko jedno dziecko o etykiecie $\Gamma \vdash \bot$.

Osąd ma dowód, jeśli istnieje poprawne drzewo dowodu, którego korzeń jest etykietowany tym osądem. Osądy mające dowód nazywamy *twierdzeniami*.

Zaproponuj typ danych reprezentujący twierdzenia w rozważanej logice i uzupełnij jego definicję w pliku `logic.ml`. Czy powinieneś ją wpisać również w pliku `logic.mli`? Przypomnij sobie pojęcie *abstrakcji danych* i zastanów się jak zapewnić, że nie da się skonstruować niepoprawnego dowodu. Czy reprezentacja twierdzeń powinna zawierać informację o całym drzewie dowodu?

Dodatkowo zaimplementuj funkcje `assumptions` oraz `consequence` zwracające odpowiednio założenia i tezę twierdzenia. Pozwolą one na zarejestrowanie funkcji drukującej twierdzenia w interpreterze.

```
utop # #install_printer Logic.pp_print_theorem ;;
```

Zadanie 6 (4p). Zaimplementuj funkcje `by_assumption`, `imp_i`, `imp_e` oraz `bot_e` z modułu `Logic` odpowiadające regułom wnioskowania. Ich dokładną specyfikację znajdziesz w pliku `logic.mli`

Zadanie 7 (2p). Korzystając z modułu `Logic` zbuduj dowody następujących twierdzeń:

- $\vdash p \to p$,
- $\vdash p \to q \to p$,
- $\vdash (p \to q \to r) \to (p \to q) \to p \to r$,
- $\vdash \bot \to p$.

W następnym odcinku ...

Moduł `Logic` dostarcza metody konstruowania dowodów *w przód*, tzn. takich gdzie z prostych znanych faktów buduje się bardziej skomplikowane. Niestety nie jest to najwygodniejsza metoda przeprowadzania dowodów w logice intuicjonistycznej. Np. skonstruowanie dowodu twierdzenia

$$\vdash (((p \to \bot) \to p) \to p) \to ((p \to \bot) \to \bot) \to p$$

nie jest łatwe, kiedy się nie wie co się robi (możesz spróbować). Znacznie łatwiej jest konstruować dowody *w tył*, tzn. upraszczać cel do udowodnienia tak długo, aż dojdziemy do rzeczy trywialnych. W terminach drzew dowodów oznacza to konstruowanie drzewa dowodu od korzenia do liści.

Celem drugiej części mini-projektu będzie zbudowanie biblioteki pozwalającej dowodzić *w tył*. Implementacja będzie znacznie bardziej skomplikowana, więc odpowiedni poziom zaufania do udowodnionych twierdzeń uzyskamy innymi środkami: odpowiedzialność za poprawność dowodu zrzucimy na moduł `Logic` z tej listy zadań. Dlatego postaraj się rozwiązać tą listę zadań, choćby już po ćwiczeniach. Jeśli jednak nie uda się Tobie tego osiągnąć, nie przejmuj się — zadania z następnej listy dalej będzie dało się rozwiązać, ale ich dobre przetestowanie może okazać się problematyczne.