Programowanie Funkcyjne 2024

Lista zadań nr 4

Na zajęcia 5 i 7 listopada 2024

Zadanie 1 (1p). Zaimplementuj listy o dostępie swobodnym oparte na systemie binarnym z nadmiarową cyfrą 2. Zademonstruj ich przewagę nad listami zaimplementowanymi na wykładzie.

Zadanie 2 (4p). W tym zadaniu będziemy rozważać strukturę danych (typ `'a zlist`) do nawigowania po listach, czyli listę z wyróżnionym kursorem (tak, będzie to zdegenerowany zipper). Zaimplementuj następujące operację na takich listach:

- `of_list: 'a list -> 'a zlist` — utworzenie nowej listy z kursorem ustawionym na początek;
- `to_list: 'a zlist -> 'a list` — konwersja do zwykłych list, zapominająca o pozycji kursora;
- `elem: 'a zlist -> 'a option` — pierwszy element za kursorem (lub `None` jeśli kursor jest na końcu);
- `move_left: 'a zlist -> 'a zlist` — przesunięcie kursora w lewo;
- `move_right: 'a zlist -> 'a zlist` — przesunięcie kursora w prawo;
- `insert: 'a -> 'a zlist -> 'a zlist` — wstawienie nowego elementu i przesunięcie kursora za ten element;
- `remove: 'a zlist -> 'a zlist` — usunięcie elementu przed kursorem.

Wszystkie operacje oprócz `to_list` powinny działać w czasie stałym!

Mini-Projekt: Asystent Dowodzenia, cz. II

Moduł `Logic` z poprzedniej listy zadań 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. Znacznie łatwiej jest konstruować dowody w tył, tzn. upraszczać cel do udowodnienia tak długo, aż dojdziemy do rzeczy trywialnych. W terminach drzew wyprowadzenia dowodu oznacza to konstruowanie drzewa od korzenia do liści.

W tym i kolejnych zadaniach będziemy rozbudowywać moduł `Proof` naszej biblioteki do logiki intuicjonistycznej. Kluczowym elementem tego modułu jest typ `proof`, który reprezentuje częściowo skonstruowany dowód w tył. Taki potencjalnie niedokończony dowód w tył jest albo kompletnym drzewem dowodu, albo drzewem dowodu w którym brakuje niektórych poddrzew, tzn. mamy dodatkowy typ liści, który reprezentuje dziurę (zwyczajowo zwaną *celem*) w dowodzie. Dodatkowo, jeden z celów jest wyróżniony (będziemy nazywać go *aktywnym*): jest to miejsce na którym się skupiamy i właśnie tam będziemy wypełniać brakujący kawałek dowodu.

Każdy z celów powinien zawierać informację o następujących elementach:

- listę (typu `(string * formula) list`) nazwanych założeń, które w tym miejscu są dostępne,
- oraz formułę do udowodnienia.

Oczekujemy, że gdy dany cel zostanie w końcu wypełniony kompletnym dowodem osądu $\Gamma \vdash \varphi$, to Γ będzie podzbiorem dostępnych założeń, a φ będzie formułą, którą należało w tym miejscu udowodnić.

Zadanie 3 (6p). Zaproponuj definicję typu `proof`, a następnie zaimplementuj trzy podstawowe funkcje operujące na dowodach:

- funkcję `proof`, która rozpoczyna (pusty) dowód podanego osądu — ten jedyny aktywny cel, to podany przez użytkownika osąd;
- funkcję `goal`, która zwraca informacje o aktywnym celu lub `None` gdy dowód jest kompletny;
- funkcję `qed`, która zamienia kompletny dowód na twierdzenie (lub zgłasza wyjątek, gdy dowód ma jeszcze dziury).

Jeśli to zadanie okaże się za trudne, to przeczytaj poniższe wskazówki (a jak jesteś odważny, to przeczytaj je dopiero po zrobieniu zadania).

- Najpierw zdefiniuj typ opisujący potencjalnie niekompletne dowody, ale bez wyróżnionego celu. Zauważ, że ten typ będzie miał jeden konstruktor opisujący cel do udowodnienia oraz po jednym konstruktorze dla każdej reguły dowodzenia.
- Warto dodać jeszcze osobny konstruktor dla kompletnych poddrzew dowodu, czyli takich które nie zawierają dziur — ułatwi to implementację następnych zadań. Kompletne poddrzewo dowodu nie musi mieć już żadnej struktury, tylko wystarczy trzymać twierdzenie (z modułu `Logic`) dowodzone przez to poddrzewo. Zauważ, że jeśli dodasz taki konstruktor, to już nie potrzebujesz osobnego konstruktora dla reguły (Ax).
- Następnie zdefiniuj typ kontekstów dla takich drzew. Przypomnij sobie zippery pokazane na wykładzie.
- Typ `proof` będzie mieć dwa konstruktory reprezentujące odpowiednio kompletny dowód i dowód z aktywnym celem. W tym drugim przypadku, by mieć wygodny dostęp do aktywnego celu, przyda się zipper: konstruktor musi pamiętać aktywny cel i kontekst w którym się on znajduje.

Zadanie 4 (3p). Zaimplementuj funkcję `next`, która cyklicznie wybiera następny cel w konstruowanym dowodzie.

Wskazówka: najpierw zaimplementuj dwie funkcje, które dla danego miejsca w dowodzie (kontekst + niedokończony dowód) szukają następnego celu idąc odpowiednio w górę i w dół drzewa.

Zadanie 5 (2p). Zaimplementuj funkcję `intro`, która w wyróżnionej dziurze próbuje dobudować kawałek drzewa odpowiadający regule $(\to\mathrm{I})$. Argument typu `string` opisuje nazwę nowego założenia. Funkcja powinna zgłosić wyjątek, gdy celem do udowodnienia nie jest implikacja.

Zadanie 6 (3p). Zaimplementuj funkcje `apply`, `apply_thm` oraz `apply_assm`. Funkcja `apply` przyjmuje formułę ψ_0 postaci $\psi_1 \to \ldots \to \psi_n \to \varphi$ albo $\psi_1 \to \ldots \to \psi_n \to \bot$ (n może być równe zero) gdzie φ jest formułą do udowodnienia w wyróżnionej dziurze i dobudowuje kawałek dowodu z reguł $(\to\mathrm{E})$ oraz $(\bot\mathrm{E})$, tak że nowo-powstałe dziury wymagają udowodnienia formuł ψ_0,ψ_1,...,ψ_n. Funkcje `apply_thm` oraz `apply_assm` działają podobnie, z tą różnicą, że od razu wypełniają dziurę odpowiadającą ψ_0 odpowiednim twierdzeniem, tym przyjętym jako argument, lub zbudowanym za pomocą reguły (Ax).

Wszystkie trzy funkcję są bardzo podobne. Postaraj się nie duplikować kodu.

Wskazówka: Przydatna może się okazać funkcja, która wypełnia wyróżnioną dziurę podanym twierdzeniem. Problemem może okazać się znalezienie następnego celu, lub stwierdzenie, że już wszystkie zostały udowodnione. Spróbuj wykorzystać (być może po drobnych modyfikacjach) funkcje pomocnicze z zadnia 4. Stwierdzenie, że dowód jest kompletny będzie łatwe, gdy będziesz utrzymywać niezmiennik mówiący o tym, że kawałek dowodu jest reprezentowany jako twierdzenie wtedy i tylko wtedy gdy nie zawiera dziur. Zapoznaj się również z funkcją assoc z modułu `List`.

Zadanie 7 (1p). Jeśli rozwiązałeś wszystkie poprzednie zadania, to zbudowałeś prostego, interaktywnego asystenta dowodzenia. Można za jego pomocą budować dowody skomplikowanych twierdzeń w prostej logice intuicjonistycznej. Sprawdź, jak się zachowuje poniższy dowód, gdy dopisujemy do niego kolejne wiersze (po uprzednim zarejestrowaniu funkcji drukujących dowody i twierdzenia).

```
proof [] {Twoja reprezentacja formuły p → (p → q) → q}
  |> intro "H1"
  |> intro "H2"
```

```
    |> apply_assm "H2"
    |> apply_assm "H1"
  |> qed
```

Następnie udowodnij poniższe twierdzenia.

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