Programowanie Funkcyjne 2020

Lista zadań nr 5 dla grup mabi/mbu/ppo/efes

Na zajęcia 10 i 18 listopada 2020

Dowodzenie w tył w intuicjonistycznym rachunku zdań

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 tych zadaniach będziemy rozbudowywać moduł `Proof` naszej biblioteki, który — budując na module `Logic` — dostarczy metod dowodzenia w tył.

Zadanie 1 (2p). Jak wspomnieliśmy wyżej, dowodzenie w tył polega na budowaniu drzewa dowodu „od korzenia". To oznacza że dopóki nie zostanie zakończony, dowód jest tylko *częściowy* i zawiera jedną lub więcej „dziur", zwanych zwyczajowo *celami*. Cele te reprezentujemy przez typ `goalDesc`, składający się z listy dostępnych, nazwanych założeń (kontekstu) i formuły którą należy udowodnić. Zaproponuj reprezentację częściowego dowodu jako definicję typu `proof`.

Warto przy tym zadbać żeby częściowy dowód przechowywał przydatne metadane (zastanów się jakie!) i zachowywał odpowiednie niezmienniki — w szczególności, dobrze reprezentować poddrzewa które są już zakończone (nie mają dziur) jako twierdzenia z modułu `Logic`. Przydatne mogą okazać się tzw. *sprytne konstruktory* (smart constructors), czyli funkcje których używamy zamiast konstruktorów, a które same dbają o zachowanie niezmienników, propagację metadanych itp.

Zdefiniowawszy typ `proof` zaimplementuj procedury `numGoals` zwracającą liczbę celów w naszym dowodzie, `goals` zwracającą listę opisów tych celów i `qed`, która kończy dowód (w którym nie ma już żadnych dziur) i tworzy z niego twierdzenie.

Zadanie 2 (2p). Z niedokończonym dowodem najwygodniej pracować z perspektywy którejś z dziur pozostałych w nim: taki dowód skoncentrowany na wyróżnionej dziurze reprezentujemy przez typ `goal`. Zaproponuj jego definicję — najwygodniej będzie, jeśli będzie ona podobna do *zippera* z zadania 2. na liście 4.

Następnie zdefiniuj funkcje `proof`, która tworzy pusty dowód i ustawia fokus w jego jedynym podcelu, `goal`, która zwraca opis aktualnie rozważanego celu, a także `focus` i `unfocus` pozwalające odpowiednio przejść do wybranego celu w częściowym dowodzie i odtworzyć dowód z konkretnego celu.

Zadanie 3 (2p). Zaimplementuj funkcje `next` i `prev`, które dla danego celu cyklicznie wybierają odpowiednio następny lub poprzedni cel w konstruowanym dowodzie.

Zadanie 4 (1p). 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 5 (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

($\perp$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. Zapoznaj się również z funkcją assoc z modułu `List`.

Zadanie 6 (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.

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

Intuicjonistyczna logika pierwszego rzędu

W kolejnych zadaniach będziemy rozbudowywać nasz system dowodzenia w kierunku intuicjonistycznej logiki pierwszego rzędu. Kluczową różnicą między rachunkiem zdań a rachunkiem pierwszego rzędu jest zastąpienie zmiennych zdaniowych w formułach przez *predykaty*, które operują na *termach*. Dla przykładu, zamiast dowodzić twierdzenia postaci:

$$\vdash p \wedge q \to q \wedge p$$

będziemy chcieli zająć się twierdzeniami postaci:

$$\vdash (\forall x.p(x) \to q(x)) \wedge (\exists y.\ p(y+z)) \to \exists y.\ q(y+z),$$

przy czym oczywiście, podobnie jak w przypadku rachunku zdań, uprościmy sobie sprawę rozważając minimalny interesujący fragment.[1]

Zadanie 7 (2p). Zanim odpowiednio przebudujemy definicję formuł naszej logiki, musimy zdefiniować czym są termy. Dla nas będą one typem danych zdefiniowanym następująco:

- każda zmienna x jest termem;
- jeżeli f jest symbolem funkcyjnym, a t_1,...,t_n są termami (n może być równe 0), to $f(t_1,\ldots,t_n)$ też jest termem.

Zazwyczaj przyjmuje się pewną ustaloną *sygnaturę* mówiącą jakie symbole funkcyjne mamy do dyspozycji i ilu każdy z nich oczekuje argumentów — np. że mamy dwuargumentowy symbol $+$ i stałe (zeroargumentowe symbole) 0 i 1. Dla uproszczenia, nie będziemy się tym przejmować i przyjmiemy że każdy symbol jest dobry i może mieć dowolnie wiele argumentów.

Aby użyć termów w naszych formułach, potrzebujemy formuł atomowych postaci $p(t_1,\ldots,t_n)$. Podobnie jak w przypadku symboli funkcyjnych, zazwyczaj pracujemy z ustalonym zbiorem symboli relacyjnych (np. binarne symbole $=$ i $<$) — ale tu znów przyjmiemy że dowolny symbol relacyjny p z dowolną ilością termów jako argumentami jest dobrą formułą atomową. Zauważmy też, że nasze formuły atomowe są co najmniej tak silne jak zmienne zdaniowe: zawsze możemy zamiast zmiennej zdaniowej p użyć zeroargumentowego symbolu relacyjnego $p()$.

[1] Jeśli potrzebujesz głębszego odświeżenia materiału, rachunek pierwszego rzędu jest przedmiotem rozdziału 3. whitebooka.

Zaproponuj definicję typu `term`, reprezentującego termy i dodaj jego implementację do plików `logic.mli` i `logic.ml`, a następnie zastąp zmienne zdaniowe w definicji typu `formula` przez ogólniejsze formuły atomowe. W typach tych symbole funkcyjne i relacyjne wygodnie reprezentować przy pomocy napisów. Następnie zmień resztę implementacji tak żeby działała ze zmodyfikowanym typem `formula`. Jeśli dobrze rozwiązałeś poprzednie zadania, to jedyne co powinieneś zrobić, to rozbudować definicję funkcji `string_of_formula` (i ewentualnie przypadków dopasowania wzorca kończących się błędem).

Zadanie 8 (2p). Najistotniejsza z punktu widzenia programistycznego różnica między rachunkiem zdań a logiką pierwszego rzędu dotyczy kwantyfikatorów — a konkretniej tego że *wiążą* one zmienne. Zauważ, że formuły $\forall x.\ p(x, y)$ i $\forall z.\ p(z, y)$ uważamy za nierozróżnialne mimo różnicy nazw: kwantyfikator *wiąże* nazwę x (lub z) w podformule, a dbamy jedynie o to, żeby wystąpienia zmiennych odnosiły się do właściwych kwantyfikatorów, nie o to jak brzmią.[2] Oznacza to że nie możemy, jak do tej pory, porównywać formuł przy użyciu operatora = — zamiast niego musimy zdefiniować osobną operację.

Rozszerz typ `formula` o kwantyfikatory uniwersalne, czyli formuły postaci $\forall x.\ \varphi$, gdzie x jest zmienną, a φ formułą. Dodaj do pliku `logic.mli` sygnaturę funkcji

```
val equal_formula : formula -> formula -> bool
```

a w pliku `logic.ml` dostarcz jej implementację. Następnie popraw resztę kodu tak, by uwzględniała tę zmianę. Nie zapomnij o zastąpieniu równości strukturalnej na nowo zdefiniowaną, również w pliku `proof.ml`.

Zadanie 9 (2p). Formuły postaci $\forall x.\ \varphi$ mówią coś o dowolnych obiektach (termach) reprezentowanych przez x — żeby użyć założenia tej postaci, musimy umieć *podstawić* w formule φ term za zmienną. Przykładowo, jeśli chcielibyśmy użyć założenia że $\forall x.\ p(+(x, x))$ dla termu $+(y, 0())$, chcielibyśmy otrzymać formułę $p(+(+(y, 0()), +(y, 0())))$.[3] W tym zadaniu zajmiemy się definicją podstawienia.

Dodaj to pliku `logic.mli` sygnatury następujących funkcji.

```
val free_in_term    : var -> term -> bool
val free_in_formula : var -> formula -> bool
val subst           : var -> term -> formula -> formula
```

Pierwsze dwie sprawdzają czy zmienna ma wolne wystąpienie odpowiednio w termie i w formule, zaś ostatnia podstawia term za zmienną wewnątrz formuły. Zaimplementuj te funkcje. Uważaj na problem *przechwycenia zmiennych*: rozważmy dwie równoważne formuły: $\forall x.\ p(z)$ i $\forall y.\ p(z)$, w których za zmienną z podstawiamy term $+(x, x)$. Jeśli zaimplementujemy podstawienie naiwnie, w pierwszym przypadku dostaniemy $\forall x.\ p(+(x, x))$ a w drugim $\forall y.\ p(+(x, x))$. Oczywiście, pierwszy z tych przypadków jest błędny, i musimy go uniknąć. W tym celu, możemy posłużyć się następującymi równościami (w których $\mathsf{fv}(m)$ oznacza zbiór zmiennych wolnych w m a $\varphi[x/t]$ — podstawienie t za x w formule φ).

$$
\begin{aligned}
(\forall y.\varphi)[x/t] &= \forall y.\varphi && \text{jeśli } x \notin \mathsf{fv}(\forall y.\varphi)\\
(\forall y.\varphi)[x/t] &= \forall y.\varphi[x/t] && \text{jeśli } y \notin \mathsf{fv}(t)\\
(\forall y.\varphi)[x/t] &= \forall z.\varphi[y/z][x/t] && \text{jeśli } z \notin \mathsf{fv}(\forall y.\varphi) \cup \mathsf{fv}(t)
\end{aligned}
$$

Zwróć uwagę, że ostatni przypadek pozwala nam uniknąć problemu w przykładzie rozważanym wyżej: możemy zmienić nazwę skwantyfikowanej zmiennej z x na y (na przykład) i ostatecznie otrzymać właściwy wynik. Musimy jednak umieć wygenerować nazwę zmiennej, która będzie odpowiednio bezpieczna, *świeża*. Jeśli używasz typu `string` do reprezentowania zmiennych możesz w tym celu użyć następującej funkcji, w której świeżość reprezentuje predykat `is_fresh`:

```
let gen_fresh is_fresh x =
  let rec base_len n =
```

[2] Tak samo jak z nazwami zmiennych w programach: funkcje `fun x -> x` i `fun y -> y` są dla nas identyczne, mimo że używają innych nazw dla argumentu.

[3] Symbol funkcyjny $+$ zapisaliśmy tym razem prefiksowo, jak każdy inny symbol funkcyjny, a stałą 0 — jako symbol funkcyjny z zerem argumentów.

```
    if n = 0 || x.[n-1] < '0' || x.[n-1] > '9' then n
    else base_len (n-1)
  in
  let base = String.sub x 0 (base_len (String.length x)) in
  let base = if base = "" then "x" else base in
  if is_fresh base then base
  else
    let rec loop n =
      let name = base ^ string_of_int n in
      if is_fresh name then name
      else loop (n+1)
    in loop 0
```

Zadanie 10 (2p). Reguły wprowadzania i eliminacji kwantyfikatora uniwersalnego wyglądają następująco.

$$\frac{\Gamma \vdash \varphi \quad x \notin \mathsf{fv}(\Gamma)}{\Gamma \vdash \forall x.\varphi}(\forall\mathrm{I}) \qquad\qquad \frac{\Gamma \vdash \forall x.\varphi}{\Gamma \vdash \varphi\{t/x\}}(\forall\mathrm{E})$$

Dodaj funkcje o sygnaturach

```
val : all_i : var -> theorem -> theorem
val : all_e : theorem -> term -> theorem
```

do modułu `Logic` odpowiadające tym regułom.

Zadanie 11 (2p). Jeśli rozwiązałeś zadania 4 lub zadanie 5 o dowodzeniu w tył, to rozszerz funkcjonalność zaimplementowanych funkcji o nowe reguły. Jeśli reprezentujesz zmienne za pomocą typu `string`, to pierwszy parametr funkcji `intro` może oznaczać nazwę wprowadzanej zmiennej lub założenia w zależności od kontekstu użycia. Alternatywnie możesz też zdefiniować nową funkcję do wprowadzania kwantyfikatora uniwersalnego. Funkcje z rodziny `apply` mogą przyjmować dodatkowy parametr (być może opcjonalny?) będący listą termów jakie należy podstawić, gdy eliminujemy kolejne kwantyfikatory.

Zadanie 12 (1p). Udowodnij kilka ciekawych twierdzeń intuicjonistycznej logiki pierwszego rzędu.