Ćwiczenia z Agdy – część pierwsza

Wojciech Jedynak Piotr Polesiuk

20 grudnia 2013

1 Uwaga

Niniejszy dokument stanowi pierwszą część zadań z Agdy. Rozwiązując wszystkie 9 zadań można uzyskać zaliczenie jednej listy seminaryjnej. Wkrótce pojawi się lista druga (za drugi punkt), która będzie składać się z jednego dużego (fascynującego!) zadania.

Lista zadań jest dostępna w dwu wariantach .pdf i .lagda. Wersję .lagda można otworzyć w edytorze tekstu i uzupełniać brakujące fragmenty bez przepisywania wszystkiego. Rozwiązania części pierwszej przyjmujemy do **20 stycznia 2014 roku**. Rozwiązania w postaci uzupełnionego pliku .lagda należy wysyłać e-mailem na adres `wjedynak@gmail.com` lub `piotr.polesiuk@gmail.com`. Zachęcamy także do zadawania pytań!

```
module Exercises where
```

2 Podstawy Izomorfizmu Curry'ego-Howarda

Fałsz zdefiniowaliśmy w Agdzie jako typ pusty:

```
data ⊥ : Set where
⊥-elim : {A : Set} → ⊥ → A
⊥-elim ()
```

Możemy teraz wyrazić negację w standardowy sposób: jako funkcję w zbiór pusty.

```
¬_ : Set → Set
¬ A = A → ⊥
```

Zadanie 1 *Udowodnij, że $p \Rightarrow \neg\neg p$, czyli dokończ poniższą definicję:*

```
pnnp : {A : Set} → A → ¬ ¬ A
pnnp = {!!}
```

Czy potrafisz udowodnić implikację w drugą stronę?

Zadanie 2 *Zdefiniuj koniunkcję jako polimorficzny typ i udowodnij reguły eliminacji oraz prawo przemienności tj. zdefiniuj typ $A \wedge B$ oraz funkcje fst, snd i swap:*

```
  -- replace the next line with the proper definition
postulate _∧_ : Set → Set → Set
fst : {A B : Set} → A ∧ B → A
fst = {!!}
snd : {A B : Set} → A ∧ B → B
snd = {!!}
swap : {A B : Set} → A ∧ B → B ∧ A
swap = {!!}
```

Zadanie 3 *Korzystając z koniunkcji z poprzedniego zadania i alternatywy z wykładu, sformułuj i spróbuj udowodnić prawa De Morgana znane z logiki klasycznej. Które z nich zachodzą w logice kostruktywnej?*

3 Liczby naturalne

Na wykładzie zdefiniowaliśmy liczby naturalne z dodawaniem następująco:

```
data ℕ : Set where
  zero : ℕ
  suc : ℕ → ℕ
{-# BUILTIN NATURAL ℕ #-}
{-# BUILTIN ZERO zero #-}
{-# BUILTIN SUC suc #-}
infix 6 _+_
_+_ : ℕ → ℕ → ℕ
zero + m = m
suc n + m = suc (n + m)
```

Zadanie 4 *Przypomnijmy definicję równości:*

```
infix 5 _≡_
data _≡_ {A : Set} (a : A) : A → Set where
  refl : a ≡ a
```

Pamiętając, że wg Izomorfizmu Curry'ego-Howarda indukcja to rekursja, udowodnij następujące własności dodawania:

```
plus-right-zero : (n : ℕ) → n + 0 ≡ n
plus-right-zero = {!!}
```

```
plus-suc-n-m : (n m : ℕ) → suc (n + m) ≡ n + suc m
plus-suc-n-m = {!!}
```

Zadanie 5 *Korzystając z poprzedniego zadania, udowodnij przemienność dodawania:*

```
plus-commutative : (n m : ℕ) → n + m ≡ m + n
plus-commutative = {!!}
```

4 Wektory

Przypomnijmy definicję wektorów:

```
data Vec (A : Set) : ℕ → Set where
  [] : Vec A 0
  _::_ : {n : ℕ} → (x : A) → (xs : Vec A n) → Vec A (suc n)
```

Zdefiniowaliśmy już m.in. konkatenację wektorów:

```
_++_ : {A : Set} → {n m : ℕ} → Vec A n → Vec A m → Vec A (n + m)
[] ++ v2 = v2
(x :: v1) ++ v2 = x :: (v1 ++ v2)
```

Zadanie 6 *Zaprogramuj funkcję vmap, która jest wektorowym odpowiednikiem map dla list. Jaka powinna być długość wynikowego wektora?*

Zadanie 7 *W Haskellu bardzo często używamy funkcji zip, która jest zdefiniowana następująco:*

```
zip :: [a] -> [b] -> [(a,b)]
zip (x:xs) (y:ys) = (x,y) : zip xs ys
zip _ _ = []
```

Jak widać, przyjęto tutaj, że jeśli listy są różnej długości, to dłuższa lista jest ucinana. Nie zawsze takie rozwiązanie jest satysfakcjonujące. Wymyśl taką sygnaturę dla funkcji zip na wektorach, aby nie dopuścić (statycznie, za pomocą systemu typów) do niebezpiecznych wywołań.

Zadanie 8 *Zaprogramuj* ***wydajną*** *funkcję odwracającą wektor. Użyj funkcji subst z wykładu, jeśli będziesz chciał zmusić Agdę do stosowania praw arytmetyki.*

Zadanie 9 *Rozważmy funkcję filter na wektorach. Jaka powinna być długość wektora wynikowego? Długość ta zależy od zadanego predykatu i samego wektora ... Możliwe są trzy podejścia:*

1. *zwrócić listę zamiast wektora,*

2. *ukryć długość wektora używając typu egzystencjalnego* ($\Sigma\ \mathbb{N}\ (\lambda\ n \to Vec\ A\ n)$)*,*

3. *napisać pomocniczą funkcję obliczającą długość wynikowego wektora.*

Zaimplementuj wszystkie trzy warianty. W trzecim wariancie użyj następujących sygnatur:

```
data Bool : Set where
  true false : Bool
filter-length : {A : Set} {n : ℕ} → (A → Bool) → Vec A n → ℕ
filter-length = {!!}
filter₃ : {A : Set} {n : ℕ} → (P : A → Bool) → (xs : Vec A n) → Vec A (filter-length P xs)
filter₃ = {!!}
```