Ćwiczenia z Agdy - Lista 1.

Wojciech Jedynak Paweł Wieczorek

18 października 2011

```
module Exercises where
```

1 Typ identycznościowy, czyli równość w języku

```
data ⊥ : Set where

⊥-elim : {A : Set} → ⊥ → A
⊥-elim ()

¬_ : Set → Set
¬ A = A → ⊥

infix 5 _≡_

data _≡_ {A : Set} : A → A → Set where
  refl : {a : A} → a ≡ a

_≢_ : {A : Set} → A → A → Set
a ≢ b = ¬ (a ≡ b)
```

2 Wartości boolowskie

Wartości boolowskie zdefiniowaliśmy następująco:

```
data Bool : Set where
  false : Bool
  true : Bool
```

Zadanie 1 *Uzupełnij poniższą definicję tak, aby otrzymać funkcję negacji boolowskiej.*

```
not : Bool → Bool
not false = {!!}
not true = {!!}
```

Udowodnij, że Twoja definicja spełnia następujące własności:

```
not-has-no-fixed-points : (b : Bool) → not b ≢ b
not-has-no-fixed-points = {!!}
not-is-involutive : (b : Bool) → not (not b) ≡ b
not-is-involutive = {!!}
```

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 2 *Pamiętając, że wg ICH indukcja = 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 3 *Korzystając z poprzedniego zadania, udowodnij przemienność dodawania:*

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

Zadanie 4 *Zdefiniuj mnożenie i potęgowanie dla liczb naturalnych.*

```
infix 7 _*_
infix 8 _^_
_*_ : ℕ → ℕ → ℕ
n * m = {!!}
_^_ : ℕ → ℕ → ℕ
n ↑ m = {!!}   -- tutaj też ma być daszek :-)
```

Jeśli masz ochotę, to udowodnij przemienność mnożenia.

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 5 *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 niedopuścić (statycznie, za pomocą systemu typów) do niebezpiecznych wywołań.

Zadanie 6 *Zaprogramuj wydajną funkcję odwracającą wektor. Użyj funkcji subst, jeśli będziesz chcieć zmusić Agdę do stosowania praw arytmetyki.*

5 Zbiory skończone - typ fin

Przypomnijmy definicję typu fin:

```
data Fin : ℕ → Set where
  zero : {n : ℕ} → Fin (suc n)
  suc  : {n : ℕ} → (i : Fin n) → Fin (suc n)
```

Dla dowolnego $n \in \mathbb{N}$ typ Fin n ma dokładnie n mieszkańców (w szczególności Fin 0 jest pusty). Własność ta sprawia, że typ Fin świetnie nadaje się do indeksowania wektorów. Indeksowanie to jest bezpieczne, gdyż system typów wyklucza nam możliwość stworzenia indeksu, który wykraczałby poza dozwolony zakres.

```
_!_ : {A : Set} {n : ℕ} → Vec A n → Fin n → A
[] ! ()
(x :: xs) ! zero = x
(x :: xs) ! suc i = xs ! i
```