Univerzitet u Beogradu
Matematički fakultet
Katedra za računarstvo i informatiku

Šifra predmeta: R265
22.04.2024.

Uvod u interaktivno dokazivanje teorema

Vežbe 09

Zadatak 1 *Tip: list.*

Diskutovati o sledećim termovima i vrednostima.

```
term []
term 1 # 2 # []
term (1::nat) # 2 # []
term [1, 2]
term [1::nat, 2]

value [1..5]
value [1..<5]

term sum-list
value sum-list [1..<5]

term map
term λ x. f x
value map (λ x. x^2) [1..<5]

value sum-list (map (λ x. x^2) [1..<5])
value ∑ x ← [1..<5]. x^2

term fold
term foldr
term foldl
value fold (λ x acc. x^2 + acc) [1..<5] (0::nat)
value foldr (λ x acc. x^2 + acc) [1..<5] (0::nat)
value foldl (λ acc x. x^2 + acc) (0::nat) [1..<5]

term filter
value filter (λ x. x > 2) [1..<5]
```

Zadatak 2 *Algebarski tip podataka: lista.*

Definisati polimorfan algebarski tip podataka *'a lista* koji predstavlja listu elemenata polimorfong tipa *'a*.

```
datatype 'a lista = Prazna
                  | Dodaj 'a 'a lista

term Dodaj (1::nat) (Dodaj 2 (Dodaj 3 Prazna))
```

Definisati funkcije *duzina'* :: *'a lista* ⇒ *nat*, *nadovezi'* :: *'a lista* ⇒ *'a lista* ⇒ *'a lista*, *obrni'* :: *'a lista* ⇒ *'a lista* primitivnom rekurzijom koje računaju dužinu liste, nadoveziju i obrću liste tipa *'a lista*.

primrec *duzina'* :: *'a lista* ⇒ *nat* **where**
  duzina' Prazna = *0*
| *duzina'* (*Dodaj - xs*) = *1* + *duzina' xs*

primrec *nadovezi'* :: *'a lista* ⇒ *'a lista* ⇒ *'a lista* **where**
  nadovezi' Prazna ys = *ys*
| *nadovezi'* (*Dodaj x xs*) *ys* = *Dodaj x* (*nadovezi' xs ys*)

primrec *obrni'* :: *'a lista* ⇒ *'a lista* **where**
  obrni' Prazna = *Prazna*
| *obrni'* (*Dodaj x xs*) = *nadovezi'* (*obrni' xs*) (*Dodaj x Prazna*)

Zadatak 3 *Osnovne funkcije nad listama.*

Definisati funkciju *duzina* :: *'a list* ⇒ *nat* primitivnom rekurzijom koja računa dužinu liste tipa *'a list*. Pokazati da su *duzina* i *length* ekvivalentne funkcije.

primrec *duzina* :: *'a list* ⇒ *nat* **where**
  duzina [] = *0*
| *duzina* (*x* # *xs*) = *1* + *duzina xs*

lemma *duzina-length*:
  shows *duzina xs* = *length xs*
  by (*induction xs*) *auto*

Definisati funkciju *prebroj* :: (*'a*::*equal*) ⇒ *'a list* ⇒ *nat* primitivnom rekurzijom koja računa koliko se puta javlja element tipa *'a*::*equal* u listi tipa (*'a*::*equal*) *list*. Pokazati da je *prebroj a xs* ≤ *length xs*.

primrec *prebroj* :: (*'a*::*equal*) ⇒ *'a list* ⇒ *nat* **where**
  prebroj a []= *0*
| *prebroj a* (*x* # *xs*)= (*if a* = *x then 1* + *prebroj a xs else prebroj a xs*)

lemma *prebroj a xs* ≤ *length xs*
  by (*induction xs*) *auto*

term *count-list*

Definisati funkciju *sadrzi* :: (*'a*::*equal*) ⇒ *'a list* ⇒ *bool* primitivnom rekurzijom koja ispituje da li se element tipa *'a*::*equal* javlja u listi tipa (*'a*::*equal*) *list*. Pokazati da je *sadrzi a xs* = *a* ∈ *set xs*

primrec *sadrzi* :: (*'a*::*equal*) ⇒ *'a list* ⇒ *bool* **where**
  sadrzi a [] ⟷ *False*
| *sadrzi a* (*x* # *xs*) ⟷ *a* = *x* ∨ *sadrzi a xs*

lemma *sadrzi a xs* ⟷ *a* ∈ *set xs*
  by (*induction xs*) *auto*

Definisati funkciju *skup* :: *'a list* ⇒ *'a set* primitivnom rekurzijom koja vraća skup tipa *'a set* koji je sačinjen od elemenata liste tipa *'a list*. Pokazati da je *skup xs* = *set xs*.

primrec *skup* :: *'a list* ⇒ *'a set* **where**
  skup [] = {}
| *skup* (*x* # *xs*) = {*x*} ∪ *skup xs*

lemma *skup xs = set xs*
  by (*induction xs*) *auto*

Definisati funkciju *nadovezi* :: *'a list ⇒ 'a list ⇒ 'a list* primitivnom rekurzijom koja nadovezuje jednu listu na drugu tipa *'a list*. Pokazati da je ekvivalentna ugrađenoj funkciji *append* ili infiksom operatoru @.

primrec *nadovezi* :: *'a list ⇒ 'a list ⇒ 'a list* **where**
  nadovezi [] *ys = ys*
| *nadovezi* (*x # xs*) *ys = x # nadovezi xs ys*

lemma *nadovezi-append*:
  shows *nadovezi xs ys = xs @ ys*
  by (*induction xs*) *auto*

Formulisati i pokazati da je dužina dve nedovezane liste, zbir dužina pojedinačnih listi.
Orediti i dokazati osobine za funkcije *skup* i *nadovezi*, kao i za *sadrzi* i *nadovezi*.

lemma *duzina-nadovezi*:
  shows *duzina* (*nadovezi xs ys*) = *duzina xs* + *duzina ys*
  by (*induction xs*) *auto*

lemma *skup-nadovezi*:
  shows *skup* (*nadovezi xs ys*) = *skup xs* ∪ *skup ys*
  by (*induction xs*) *auto*

lemma *sadrzi-nadovezi*:
  shows *sadrzi a* (*nadovezi xs ys*) = *sadrzi a xs* ∨ *sadrzi a ys*
  by (*induction xs*) *auto*

Zadatak 4 *Obrtanje liste.*

Definisati funkicju *obrni* :: *'a list ⇒ 'a list* primitivnom rekurzijom koja obrće listu tipa *'a list*. Pokazati da funkcija je *obrni* ekvivalentna funkciji *rev*. Nakon toga pokazati da je dvostruko obrnuta lista ekvivalentna početnoj listi.
Napomena: Pri definisanju funkcije *obrni* nije dozvoljeno koristiti operator nadovezivanje @.
Savet: Potrebno je definisati pomoćne leme.

primrec *obrni* :: *'a list ⇒ 'a list* **where**
  obrni [] = []
| *obrni* (*x # xs*) = *nadovezi* (*obrni xs*) [*x*]

lemma *obrni-rev*:
  shows *obrni xs = rev xs*
  by (*induction xs*) (*auto simp add*: *nadovezi-append*)

lemma *nadovezi-asoc*:
  shows *nadovezi* (*nadovezi xs ys*) *zs* = *nadovezi xs* (*nadovezi ys zs*)
  by (*induction xs*) *auto*

lemma *nadovezi-Nil-desno* [*simp*]:
  shows *nadovezi xs* [] = *xs*
  by (*induction xs*) *auto*

lemma *obrni-nadovezi* [*simp*]:
shows *obrni* (*nadovezi xs ys*) = *nadovezi* (*obrni ys*) (*obrni xs*)
by (*induction xs*) (*auto simp add*: *nadovezi-asoc*)

lemma *obrni-obrni-id*: *obrni* (*obrni xs*) = *xs*
by (*induction xs*) *auto*

Definisati funkciju *snoc* :: *'a* ⇒ *'a list* ⇒ *'a list* koja dodaje element na kraj liste, i funkciju *rev-snoc* :: *'a list* ⇒ *'a list* koja uz pomoć funkcije *snoc* obrće elemente liste. Da li *rev-snoc* popravlja složenost obrtanja liste?

primrec *snoc* :: *'a* ⇒ *'a list* ⇒ *'a list* **where**
snoc a [] = [*a*]
| *snoc a* (*x* # *xs*) = *x* # *snoc a xs*

primrec *rev-snoc* :: *'a list* ⇒ *'a list* **where**
rev-snoc [] = []
| *rev-snoc* (*x* # *xs*) = *snoc x* (*rev-snoc xs*)

Definisati funkciju *itrev* koja obrće listu iterativno.
Savet: Koristiti pomoćnu listu.

primrec *itrev* :: *'a list* ⇒ *'a list* ⇒ *'a list* **where**
itrev [] *ys* = *ys*
| *itrev* (*x* # *xs*) *ys* = *itrev xs* (*x* # *ys*)

Pokazati da je funkcija *itrev* ekvivalentna ugrađenoj funkciji *rev*, kada je inicijalna pomoćna lista prazna.

lemma *itrev-rev-append*:
shows *itrev xs ys* = *rev xs* @ *ys*
by (*induction xs arbitrary*: *ys*) *auto*

lemma *itrev-rev*:
shows *itrev xs* [] = *rev xs*
by (*induction xs*) (*auto simp add*: *itrev-rev-append*)

Pomoću funkcije *fold* opisati obrtanje liste. Pokazati ekvivalentnost funkciji *itrev* sa obrtanjem liste preko *fold*-a.

lemma *fold-Cons-append*:
shows *fold* (#) *xs ys* @ *zs* = *fold* (#) *xs* (*ys* @ *zs*)
by (*induction xs arbitrary*: *ys zs*) *auto*

lemma *itrev-fold-Cons*:
shows *itrev xs ys* = *fold* (#) *xs ys*
by (*induction xs arbitrary*: *ys*) (*auto simp add*: *itrev-rev-append fold-Cons-append*)

Zadatak 5 *Zamena u listi.*

Definisati funkcju *zameni* :: *'a* ⇒ *'a* ⇒ *'a list* ⇒ *'a list* primitivnom rekurzijom, tako da *zameni a b xs* u listi *xs* zamenjuje sva pojavljivanja elementa *a* sa elementom *b*. Pokazati da je funkcija *zameni* korektna preko zadatih lema.

primrec *zameni* :: *'a* ⇒ *'a* ⇒ *'a list* ⇒ *'a list* **where**
  zameni a b [] = []
| *zameni a b* (*x* # *xs*) =
      (*if a* = *x then b* # *zameni a b xs else x* # *zameni a b xs*)

lemma *zameni-length*: *length* (*zameni a b xs*) = *length xs*
  by (*induction xs*) *auto*

lemma *zameni-set*: *a* ≠ *b* ⟹ *a* ∉ *set* (*zameni a b xs*)
  by (*induction xs*) *auto*

lemma *zameni-set2*: *b* ∈ *set xs* ⟹ *b* ∈ *set* (*zameni a b xs*)
  by (*induction xs*) *auto*

Definisati funkciju *zameni'* koja u listi zamenjuje određeni element drugim elementom. Funkcija *zameni'* treba da bude repno rekurzivna.
Savet: Kao u primeru *itrev* koristiti pomoćnu listu.

primrec *zameni'* :: *'a* ⇒ *'a* ⇒ *'a list* ⇒ *'a list* ⇒ *'a list* **where**
  zameni' - - [] *ys* = *rev ys*
| *zameni' a b* (*x* # *xs*) *ys* =
    (*if a* = *x then zameni' a b xs* (*b* # *ys*) *else zameni' a b xs* (*x* # *ys*))

lemma *zameni'-len-gen*: *length* (*zameni' a b xs ys*) = *length xs* + *length ys*
  by (*induction xs arbitrary*: *ys a b*) *auto*

lemma *zameni'-len*: *length* (*zameni' a b xs* []) = *length xs*
  by (*induction xs*) (*auto simp add*: *zameni'-len-gen*)

lemma *zameni'-set-gen*: (*a* ≠ *b* ∧ *a* ∉ *set ys*) ⟹ *a* ∉ *set* (*zameni' a b xs ys*)
  by (*induction xs arbitrary*: *ys a b*) *auto*

lemma *zameni'-set*: *a* ≠ *b* ⟹ *a* ∉ *set* (*zameni' a b xs* [])
  by (*induction xs*) (*auto simp add*: *zameni'-set-gen*)

lemma *zameni'-set2-gen*: (*b* ∈ *set xs* ∨ *b* ∈ *set ys*) ⟹ *b* ∈ *set* (*zameni' a b xs ys*)
  by (*induction xs arbitrary*: *ys a b*) *auto*

lemma *zameni'-set2*: *b* ∈ *set xs* ⟹ *b* ∈ *set* (*zameni' a b xs* [])
  by (*induction xs*) (*auto simp add*: *zameni'-set2-gen*)