Računalna formalizacija matematičkih dokaza

Čižić, Ivona

SVEUČILIŠTE U ZAGREBU

PRIRODOSLOVNO–MATEMATIČKI FAKULTET

MATEMATIČKI ODSJEK

Ivona Čižić

RAČUNALNA FORMALIZACIJA MATEMATIČKIH DOKAZA

Diplomski rad

Voditelj rada:
doc. dr. sc. Vedran Čačić

Zagreb, rujan 2023.

Ovaj diplomski rad obranjen je dana _____ pred ispitnim povjerenstvom u sastavu:

1. _____, predsjednik
2. _____, član
3. _____, član

Povjerenstvo je rad ocijenilo ocjenom _____.

Potpisi članova povjerenstva:

1. _____
2. _____
3. _____

Roditeljima koji su me podupirali kroz cijelo moje obrazovanje i mentoru koji mi je pomagao u izradi ovog rada

Sadržaj

Uvod

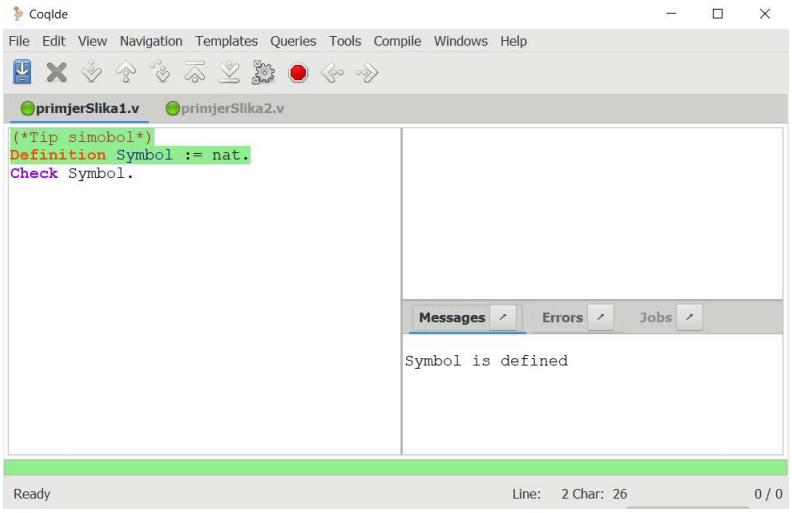
Formalne metode u matematici doživjele su procvat početkom ovog stoljeća, kako s razvojem tehnologije računala postaju prikladna za sve više primjena na matematičko rezoniranje. Cilj ovog rada je razbijanje straha od formalnih dokaza i funkcijskog programiranja.

Za moderno bavljenje matematikom sve je važnije poznavati neki od računalnih sustava za formalno dokazivanje, bilo za provjeru rezultata, strogo fundiranje matematičkih područja, ili ekstrakciju programa. Diplomski rad opisuje računalni sustav za formalno dokazivanje pod imenom Coq. Sistem Coq nudi bogato okruženje za interaktivan razvoj računalno provjerenih koraka formalnog zaključivanja. Jezgra Coqovog sistema je jednostavni kontrolor dokaza koji jamči da su izvedeni isključivo točni koraci u zaključivanju. Na tu jezgru se nastavljaju biblioteke koje sadrže niz općepoznatih definicija i lema, taktike za polu-automatsko konstruiranje složenih dokaza i posebni programski jezik koji omogućuje definiranje novih taktika za automatizaciju dokaza. Kroz rad ćemo prikazati neke od čestih područja primjene sistema Coq, kao što su realistično okruženje za funkcijsko programiranje i njegova uloga kao pomoćnika pri dokazivanju u logici višeg reda.

Poglavlje 1

Coq kao funkcijski programski jezik

Coq je jezik napravljen da bude alat pri formalizaciji matematičkih dokaza. Korisnik interaktivno uz pomoć Coqa piše željeni dokaz koristeći alate koje Coq poznaje i omogućava.



Slika 1.1: Prikaz sučelja `CoqIde`

Pokretanjem programa `CoqIde` otvaraju se tri prozora koja čine sučelje u kojem radimo. U prvi prozor pišemo kod, a u preostala dva se nalazi ispis. Gornji od njih će prikazivati trenutno stanje (*kontekst*) dokaza, dok donji prikazuje poruke o greškama, pomoć, rezultate pretraživanja i slično. Uz obični kursor za uređivanje teksta, imamo i *logički kursor* koji odvaja izvršeni (zeleni) dio od još neizvršenog.

1.1 Definiranje novih objekata u Coqu

Ukratko ćemo na manjim dijelovima koda prezentirati neke od osnovnih funkcionalnosti koje nam Coq nudi, a dobar izvor informacija za početnike je [2]. Coq barata *formulama*, provjerava jesu li dobro oblikovane i dokazuje ih, samostalno ili (češće) u suradnji s korisnikom. Formule u sebi mogu sadržavati *funkcije*, od kojih su neke unaprijed definirane — na primjer, računske operacije.

```
(*Tip simbola od kojih ćemo sastavljati riječi*)
Definition Symbol := nat.
```

Naredbu `Definition` ovdje smo koristili kako bismo definirali novi tip u Coqu. Novi tip smo nazvali `Symbol` te je po definiciji on jednak unaprijed definiranom tipu `nat`, tipu prirodnih brojeva.

```
Definition funkcija (x y : nat) := x*y + 2*x +1.
```

Sličnim postupkom kao u prethodnoj liniji, ovdje smo definirali funkciju. Iza imena funkcije napisali smo da ona prima dva argumenta, x i y tipa `nat`. Kada navodimo kojeg su tipa argumenti funkcije, koristimo znak dvotočke kako bismo odvojili ime argumenta od njegovog tipa. U konkretnom primjeru su varijable istog tipa pa im možemo zajedno navesti tip.

U najvećem broju slučajeva (pa tako i ovdje), Coq iz načina na koji koristimo argumente zna odrediti (*type inference*) njihove tipove, kao i povratni tip funkcije. Ipak, tipove je često dobro pisati radi dokumentacije.

```
Definition funkcija_2 := fun x y : nat => x*y + 2*x + 1.
```

Ovdje smo prikazali drugi način na koji možemo definirati istu funkciju. Sada smo koristili ključnu riječ `fun` kako bismo naznačili da je riječ o funkciji, a iza nje samo navodimo argumente funkcije, dok se povratna vrijednost funkcije nalazi iza znakova =>.

```
Definition konstanta := 2.
```

Naredbu `Definition` također možemo koristiti i kako bismo nekoj konstanti pridružili željenu vrijednost. U ovom primjeru pridružena je konkretna brojčana vrijednost.

```
Definition vrijednost_funkcije := funkcija 1 konstanta.
```

Konstanti `vrijednost_funkcije` je pridružena vrijednost koju dobijemo pozivom funkcije `funkcija` s argumentima 1 i prethodno definiranom konstantom. Ovdje možemo vidjeti da se argumenti koje koristimo za poziv funkcije navode nakon imena funkcije te

su odvojeni razmakom, bez korištenja zareza ili stavljanja unutar zagrada. Potrebno je razumjeti da Coq ne izračunava tu vrijednost automatski pri pridruživanju (*lazy evaluation*). Postoje naredbe odnosno taktike kojima se ta vrijednost može izračunati (*compute*) odnosno definicija raspisati (*unfold*).

```
(*Struktura za tablicu*)
Structure Tablica {A} := {
  podaci: list A;
  broj_stupaca : nat }.
```

Još jedna vrsta objekata koje možemo definirati je **struktura**, što postižemo pomoću naredbe `Structure`. Ovdje definiramo strukturu imena `Tablica` koja se sastoji od liste elemenata tipa `A` (parametar definicije) i prirodnog broja `broj_stupaca`.

1.2 Meta-naredbe

Sljedeća važna naredba u Coqu je `Check`. Naredbu `Check` koristimo kako bismo provjerili je li izraz, u ovom slučaju novodefinirana `funkcija_2`, ispravno oblikovan. To znači da se argumenti (i ostali objekti) u izrazu koriste u skladu s njihovim (deklariranim ili zaključenim) tipovima. Ako je izraz ispravno oblikovan, dobivamo povratnu poruku o tome kojeg je on tipa.

```
Check funkcija_2.
```

```
funkcija_2
 : nat -> nat -> nat
```

Funkcija je ispravno oblikovana pa nam je Coq vratio ime funkcije koje je znakom dvotočke odvojio od njenog tipa — koji se sastoji od tipa prvog argumenta, tipa drugog argumenta i tipa vrijednosti funkcije, međusobno odvojenih znakovima `->`. U poglavlju 2 objasnit ćemo značenje takvih izraza za tipove.

Ako se želimo prisjetiti definicije nekog objekta, iskoristit ćemo naredbu `Print`.

```
Print funkcija_2.
```

Ako želimo *evaluirati* (izračunati) neki izraz, koristimo naredbu `Compute`.

```
Compute funkcija_2 1 2.
```

```
funkcija_2 = fun x y : nat => x * y + 2 * x + 1
     : nat -> nat -> nat

Arguments funkcija_2 (_ _)%nat_scope
```

```
= 5
 : nat
```

Izračunata vrijednost funkcije `funkcija_2` s argumentima 1 i 2 iznosi 5 i tipa je `nat`.

```
Require Import Bool.
Require Import Arith.
```

Kako bismo mogli koristiti neke unaprijed definirane tipove u Coqu, moramo učitati biblioteku koja ih sadrži. Za tu namjenu koristimo naredbu `Require Import`. Učitane su biblioteke `Bool`, koja sadrži istinosne vrijednosti, i `Arith`, koja nam olakšava rad s prirodnim brojevima.

1.3 Konstrukti koje ćemo koristiti

```
Compute let lokalna_funkcija := fun x => x*x+2
        in lokalna_funkcija 2.
```

Jedan od konstrukata koje možemo koristiti pri pisanju formula je konstrukt `let...in`. On nam omogućava da definiramo *privremenu varijablu* koju ćemo koristiti samo u dijelu izraza nakon ključne riječi `in`.

```
Definition vrijednost_nula (x : nat) :=
  match x with
    | 0 => true
    | _ => false
  end.
```

Drugi važni konstrukt koji ćemo često koristiti je `match...with...end`. Pomoću njega možemo granati kod ovisno o obliku (*konstruktoru*) objekata nekog induktivnog tipa, kao i dobiti vrijednosti od kojih je taj objekt konstruiran. Više o induktivnim tipovima reći ćemo u točki 2.5. U koju granu funkcije `vrijednost_nula` ulazimo ovisi o vrijednosti primljenog elementa x: ako je `0` vraćamo istinu, a u suprotnom (ako je sljedbenik) laž.

```
Section Elts.
    ...
End Elts.
```

Pomoću naredbe `Section` definiramo *odjeljke* u kojima možemo koristiti lokalno deklarirane varijable pri deklaraciji ostalih objekata odjeljka. Odjeljak moramo završiti naredbom `End` nakon koje slijedi ime odjeljka.

1.4 Rekurzija pomoću naredbe (`Program`) `Fixpoint`

```
Fixpoint nth {A} (n:nat) (l:list A) (default:A) {struct l} : A
    := match n, l with
         | O, x :: l' => x
         | O, other => default
         | S m, nil => default
         | S m, x :: t => nth m t default
       end.
```

Pomoću naredbe `Fixpoint` možemo definirati **strukturnu rekurziju**. Kod takve rekurzije je važno napomenuti da funkcija koju definiramo može biti rekurzivno pozvana samo na *podizrazu* početnog argumenta. Rekurzivna funkcija `nth` prima prirodni broj `n` i listu `l` te ih koristi za `match`. Funkcija se grana u četiri grane ovisno o obliku argumenata; u zadnjoj navedenoj grani vidimo da poziva samu sebe s elementima `m` i `t`, pri čemu je `n` jednak `S m` (`S` je funkcija sljedbenik), a lista `l` je jednaka `x::t` (pri čemu je `x` prvi element liste, a `t` čini ostatak liste koji ponekad nazivamo repom liste). Uočavamo da se takav rekurzivni poziv funckije `nth` može strukturno realizirati jer je `t` podizraz od `x::t`. Također, `m` je podizraz od `S m`, ali to nam trenutno nije bitno jer smo pomoću `struct` rekli da strukturna rekurzija ide po `l`. Na taj način možemo rekurzivno proći cijelu kroz cijelu listu dok ne pronađemo njen `n`-ti element ili ne dođemo do kraja liste.

```
    (*Funkcija koja radi backtracking i vraća traženu riječ.*)
Program Fixpoint vratiRijec (t:Tablica) (r1 r2:Rijec) (i j :nat)
                {measure (i+j)} :=
match (podaci t) with
    | nil => nil
    | _ => if ( Nat.eqb (nth i r1 0) (nth j r2 0) )
           then
               match i, j with
                 | S i', S j'=> ( vratiRijec t r1 r2 i' j' )
```

```
                        ++ ((nth i r1 0)::nil)
          | _, _=> ((nth i r1 0)::nil)
        end
      else
        match i, j with
          | 0, 0 => nil
          | 0, S j' => (vratiRijec t r1 r2 i j')
          | S i', 0 => (vratiRijec t r1 r2 i' j)
          | S i', S j' =>
            if Nat.leb (ijth t i j') (ijth t i' j)
              then (vratiRijec t r1 r2 i' j)
              else (vratiRijec t r1 r2 i j')
        end
end.
```

Ovdje koristimo naredbu `Program Fixpoint` koja nam omogućava da napišemo funkciju kao u običnom funkcijskom programskom jeziku, koristeći pritom Coqovu aparaturu za dokazivanje kako bismo osigurali da je funkcija totalna, odnosno da rekurzija uvijek stane. Ova naredba nam omogućava da koristimo metodu `measure` na izrazu sastavljenom od više argumenata, što je u našem primjeru `(i+j)`. Uz to možemo promijeniti i dobro utemeljenu relaciju koju metoda `measure` koristi, a koja je podrazumijevamo postavljena na relaciju <, što nam ovdje odgovara. Korištenjem naredbe `Program` uz naredbu `Fixpoint` kompiliranjem programa može doći do stvaranja *obligacija*, tvrdnji koje je potrebno dokazati kako bi se program mogao pokrenuti.

1.5 Primjer razrješavanja obligacija

```
Next Obligation.
```

To je naredba koju ćemo koristiti kako bismo započeli razrješavati iduću obligaciju.

Njenim izvršavanjem dobivamo izlaz:

```
1 goal
t : Tablica
r1, r2 : Rijec
j', i' : nat
vratiRijec : Tablica ->
             Rijec ->
             Rijec ->
```

```
              forall i j : nat, i + j < S i' + S j' -> list Symbol
_____________________________________(1/1)
i' + j' < S i' + S j'
```

Prvi red nam govori da trenutno imamo samo jedan cilj za dokazati. Pri dokazivanju ćemo koristiti metode koje nazivamo *taktikama*. Više o taktikama reći ćemo u poglavlju 3. Svaki cilj se sastoji od dva dijela, konteksta (pretpostavki) i zaključka. Elementi konteksta su obično oblika `a:skup` ili `H:formula`, a nazivamo ih *hipotezama*. Hipoteze predstavljaju tvrdnje odnosno objekte koje možemo koristiti pri dokazivanju.

```
clear vratiRijec.
```

Na trenutni dokaz primjenjujemo taktiku `clear` koja briše zadanu hipotezu iz konteksta. U ovom slučaju obrisali smo funkciju `vratiRijec` jer nam neće biti potrebna pri dokazivanju (cilj je sasvim aritmetički). Izlaz koji dobivamo je:

```
1 goal
t : Tablica
r1, r2 : Rijec
j', i' : nat
_____________________________________(1/1)
i' + j' < S i' + S j'
```

Funkcija `vratiRijec` se sada više ne nalazi u kontekstu i ne možemo je koristiti u daljnjem dokazivanju cilja.

```
cbn.
```

To je taktika koju koristimo da bismo *normalizirali* (djelomično izračunali) izraz u zadanom cilju.

```
1 goal
t : Tablica
r1, r2 : Rijec
j', i' : nat
_____________________________________(1/1)
i' + j' < S (i' + S j')
```

Taktika `cbn` desnu stranu nejednakosti zapisuje kao sljedbenik, koristeći pravilo $S\,x + y = S\,(x + y)$ Peanove aritmetike.

```
Search S (_ + _).
```

Kako bismo pronašli već dokazane *leme*, koristimo naredbu `Search`. Uz riječ `Search` navodimo oblik izraza koji želimo pronaći, a broj rezultata dobivenih pretraživanjem ovisi o preciznosti upita i broju uključenih biblioteka. Upit iz primjera nam vraća:

```
plus_Sn_m: forall n m : nat, S n + m = S (n + m)
plus_n_Sm: forall n m : nat, S (n + m) = n + S m
mult_n_Sm: forall n m : nat, n * m + n = n * S m
```

Među dobivenim rezultatima vidimo lemu `plus_n_Sm` koja nam odgovara za provođenje sljedećeg koraka dokaza.

```
rewrite plus_n_Sm.
```

Koristili smo taktiku `rewrite` koja će podizraze zamijeniti njima ekvivalentnim izrazima koristeći pronađenu lemu.

```
1 goal
t : Tablica
r1, r2 : Rijec
j', i' : nat
______________________________________(1/1)
i' + j' < i' + S (S j')
```

Dobivamo novi cilj koji je potrebno razriješiti. Ponovno koristimo naredbu `Search` kako bismo našli sljedeću pogodnu lemu i pronalazimo je kao `plus_le_lt_compat`.

```
apply plus_le_lt_compat.
```

Taktika `apply` nam omogućuje da primijenimo željenu lemu na cilj. Lema koju primjenjujemo je `plus_le_lt_compat`:

```
forall n m p q :  nat, n <= m -> p < q -> n + p < m + q.
```

```
2 goals
t : Tablica
r1, r2 : Rijec
j', i' : nat
______________________________________(1/2)
i' <= i'
______________________________________(2/2)
j' < S (S j')
```

Primjenom prethodne taktike, dobili smo 2 cilja koja je potrebno dokazati. Da bismo se fokusirali na prvi od njih koristimo meta-taktiku crtica -. Njenim izvršavanjem dobijemo:

```
1 goal
t : Tablica
r1, r2 : Rijec
j', i' : nat
______________________________________(1/1)
i' <= i'
```

Ovaj cilj razrješavamo primjenom leme `Nat.le_refl`, koja kaže da je relacija $\leq$ na $\mathbb{N}$ refleksivna. To zajedno s meta-taktikom - pišemo (što ćemo raditi i ubuduće):

```
- apply Nat.le_refl.
```

```
This subproof is complete, but there are some unfocused goals:
______________________________________(1/1)
j' < S (S j')
```

Dovršetkom ove grane dokaza ispisuje nam se preostali cilj koji još nismo dokazali. Kao i prije, primjenom crtice se fokusiramo na cilj:

```
1 goal
t : Tablica
r1, r2 : Rijec
j', i' : nat
______________________________________(1/1)
j' < S (S j')
```

Uočavamo da su hipoteze jednake kao i u prethodnom cilju.

```
  - transitivity (S j').
```

Zadani cilj rješavamo korištenjem tranzitivnosti, precizirajući da je srednji element nejednakosti sljedbenik od `j'`.

```
2 goals
t : Tablica
r1, r2 : Rijec
j', i' : nat
```

```
______________________________________(1/2)
j' < S j'
______________________________________(2/2)
S j' < S (S j')
```

Dobili smo dva cilja koja je potrebno razriješiti. Na oba cilja možemo primijeniti istu lemu pa koristimo meta-taktiku `all:` koja nam omogućava da istu taktiku primijenimo na sve trenutno vidljive ciljeve.

```
all: apply Nat.lt_succ_diag_r.
```

Primjenom navedene naredbe dobivamo poruku da smo uspjeli razriješiti sve ciljeve.

```
No more goals.
```

Preostaje nam završiti dokaz.

```
Qed.
```

Naredba `Qed.` služi za završetak dokaza i njegovo spremanje. Na isti način kao na početku provjeravamo postoje li još neke obligacije (naredbom *Next Obligation*) i ako postoje razrješavamo ih na upravo pokazani način.

1.6 Primjer: najdulji zajednički podniz

Pomoću Coqa ćemo riješiti sljedeći problem: zadana su dva niza proizvoljnih duljina, a želimo pronaći njihov najdulji zajednički podniz. Pri tome, podniz ne mora biti uzastopan, na primjer niz $(1, 2, 3)$ je podniz niza $n_1 = (1, 1, 2, 2, 4, 4, 3, 3)$.

Rješavanje ćemo pokazati na primjeru sljedećeg para nizova:

n_1	1	1	2	2	4	4	3	3	
n_2	1	2	1	2	4	3	4	3	1

Algoritam koji ćemo koristiti zasniva se na dinamičkom programiranju: rješavamo početni problem za dva niza pomoću rješenja manjih problema za njihove početne komade. Označimo s x_i podniz niza n_1 sastavljenog od nultog do i-tog elementa u nizu n_1, za $i \in \{0, \ldots, |n_1| - 1\}$, i analogno s y_j podniz niza n_2, za $j \in \{0, \ldots, |n_2| - 1\}$. U tablici na mjestu (i, j) zabilježit ćemo duljinu najduljeg zajedničkog podniza $m_{i,j}$ za podnizove x_i i y_j. Za određivanje pojedinog elementa tablice dovoljno je promatrati njemu susjedne elemente koji mu prethode u retku, stupcu ili dijagonalno.

Pojednostavljeno, za odrediti vrijednost elementa d dovoljno je znati vrijednosti od a, b i c uz usporedbu jednakosti odgovarajućih elemenata nizova.

a	b
c	d

Algoritam koji nam omogućuje jednostavno popunjavanje tablice možemo opisati pomoću formule:

$$d = \begin{cases} \max\{b, c\}, & n_2[j] \neq n_1[i] \\ a + 1, & n_2[j] = n_1[i] \end{cases} \tag{1.1}$$

Kako bi algoritam bio ispravan, pretpostavit ćemo da će svi rubni elementi u prvom stupcu/retku imati nulu kao prethodnika u svom stupcu/retku. Opišimo malo detaljnije svaki od slučajeva koji se može pojaviti pri izvođenju algoritma.

Primjer popunjavanja tablice:

	1	2	1	**2**	4	3	4	3	1
1	1	1	1	1	1	1	1	1	1
1	1	1	2	2	2	2	2	2	2
2	1	2	2	d					
⋮									

.

Vidimo da su elementi nizova n_1, n_2 za indekse $i = 2, j = 3$ jednaki i tada ćemo element tablice na poziciji (i, j) izračunati tako da elementu tablice na poziciji $(i-1, j-1)$, prethodno označenom kao a, pribrojimo broj 1.

Druga situacija je kada imamo različite elemente u tablici:

	1	2	1	2	**4**	3	4	3	1
1	1	1	1	1	1	1	1	1	1
1	1	1	2	2	2	2	2	2	2
2	1	2	2	3	d				
⋮									

.

Elementi niza n_1, n_2 za indekse $i = 2, j = 4$ su različiti te ćemo element tablice na poziciji (i, j) kopirati iz prethodnog retka/stupca, ovisno o tome koji ima veću vrijednost. U ovom slučaju vrijednost prethodnog elementa u istom retku iznosi 3 i veća je od vrijednosti prethodnog elementa u istom stupcu koji je jednak 2 pa će novi element u tablici imati vrijednost 3.

Preostaje nam još posljedni slučaj:

	1	2	1	**2**	4	3	4	3	1
1	1	1	1	1	1	1	1	1	1
1	1	1	2	2	2	2	2	2	2
2	1	2	2	3	3	3	3	3	3
2	1	2	2	3	3	3	3	3	3
4	1	2	2	*d*					
⋮									

.

Elementi niza n_1, n_2 za indekse $i = 4$, $j = 3$ su različiti te ćemo element tablice na poziciji (i, j) kopirati analogno kao u prethodnom slučaju. U ovom slučaju vrijednost prethodnog elementa u istom retku iznosi 2 i manja je od vrijednosti prethodnog elementa u istom stupcu koji je jednak 3 pa će novi element u tablici biti vrijednosti 3.

Sada možemo popuniti cijelu tablicu:

	1	2	1	2	4	3	4	3	1
1	1	1	1	1	1	1	1	1	1
1	1	1	2	2	2	2	2	2	2
2	1	2	2	3	3	3	3	3	3
2	1	2	2	3	3	3	3	3	3
4	1	2	2	3	4	4	4	4	4
4	1	2	2	3	4	4	5	5	5
3	1	2	2	3	4	5	5	6	6
3	1	2	2	3	4	5	5	6	6

Jednom kad smo našli duljinu najduljeg zajedničkog podniza, koristimo *backtracking* kroz dobivenu tablicu kako bismo pronašli njegove elemente. Počinjemo u donjem desnom kutu gdje se nalazi najveći element tablice. Promatramo elemente niza na rubu tablice: ako su oni jednaki penjemo se jedan element dijagonalno, a ako su različiti pomičemo se jedan element gore ili lijevo, ovisno o tome koji je element tablice veći. Svaki put kada su elementi nizova jednaki, spremamo taj element na početak liste. Tako dobivena lista je najdulji zajednički podniz zadanih nizova.

		1	2	**1**	**2**	**4**	3	**4**	**3**	1
	0	0	0	0	0	0	0	0	0	0
1	0	1	1	1	1	1	1	1	1	1
1	0	1	1	2	2	2	2	2	2	2
2	0	1	2	2	3	3	3	3	3	3
2	0	1	2	2	3	3	3	3	3	3
4	0	1	2	2	3	4	4	4	4	4
4	0	1	2	2	3	4	4	5	5	5
3	0	1	2	2	3	4	5	5	6	6
3	0	1	2	2	3	4	5	5	6	6

Najdulji zajednički podniz ovih nizova je $m = (1, 1, 2, 4, 4, 3)$.

1.7 Funkcije potrebne za rješavnje problema iz primjera

Opišimo još neke funkcije koje smo koristili za implementaciju algoritma za pronalaženje najduljeg zajedničkog podniza.

Kao ulazni podatak ćemo korisiti novodefinirani objekt `Rijec` koji predstavlja listu objekata tipa `Symbol`. Prisjetimo se, `Symbol` smo definirali kao `nat`, tip prirodnih brojeva.

```
Definition Rijec := list Symbol.
```

Trebamo konstruirati tablicu. Struktura `Tablica` koju pritom koristimo pamti sve elemente tablice u jednoj listi koju smo nazvali `podaci`, tj. kao jednodimenzionalno polje, a drugi element strukture je `broj_stupaca`. Kako bismo pristupili elementu tablice na mjestu (i, j) koristimo funkciju `ijth`. Funkcija `ijth` prima tablicu t tipa `Tablica` te indekse i i j. Elementu na poziciji (i, j) možemo pristupiti ako izračunamo na kojoj poziciji u jednodimenzionalnom polju se on nalazi: `i * broj_stupaca + j`.

```
Definition ijth (t:Tablica) (i j:nat) :=
  nth (Nat.add j (Nat.mul (broj_stupaca t) i)) (podaci t) 0.
```

Pri izračunavanju idućeg elementa tablice, moramo pogledati susjedne elemente koji mu prethode. Ako se nalazimo na rubu tablice Coq neće moći pronaći prethodnika rubnog indeksa jer -1 nije tipa `nat`, a povremeno bismo htjeli nulu lijevo od lijevog ruba ili gore od gornjeg. Kako bismo to postigli definirali smo funkcije `i1jth`, `ij1th` i `i1j1th`. Ovdje navodimo jednu od njih.

```
Definition i1jth (t:Tablica)(i j:nat) :=
  match i with
    | 0 => 0
    | S i' => ijth t i' j
  end.
```

Kako bismo izračunali sljedeći element u tablici formulom (1.1), koristimo funkciju `next`. Ona prima tablicu `t`, elemente nizova koje uspoređujemo s_1 i s_2 te indekse `i` i `j` koji određuju našu poziciju u tablici.

```
(*Funkcija za idući element*)
Definition next (t:Tablica) (s1 s2:Symbol) (i j:nat) :=
  match is_equal s1 s2 with
    | true => S ( i1j1th t i j )
    | false => Nat.max (i1jth t i j) (ij1th t i j)
  end.
```

Funkcija `dodaj` uzima proslijeđenu tablicu `t` i element `x` koji je potrebno zapisati u nju te vraća novu tablicu sa zapisanim `x`. Pritom lista `podaci` nije nužno popunjena do kraja, odnosno duljina joj nije višekratnik od `broj_stupaca`. Drugim riječima, tablica postaje pravokutna tek nakon što dodamo sve elemente u nju.

```
(*Funkija za iteraciju*)
Definition dodaj (t:Tablica) (x:nat):=
  {|podaci := (podaci t) ++ (x::nil);
    broj_stupaca := broj_stupaca t;|}.
```

Funkcija `iter` prima tablicu `t` i nizove `r1` i `r2`. Njena zadaća je odrediti na kojoj poziciji se nalazimo u tablici pri njenom generiranju i dodati novi element u nju odgovarajućim pozivom funkcije `dodaj`.

```
Definition iter (t: Tablica) (r1 r2:Rijec) :=
   let i := Nat.div (length (podaci t)) (length r2) in
   let j := Nat.modulo (length (podaci t)) (length r2) in
   dodaj t (next t (nth i r1 0) (nth j r2 0) i j).
```

Definiramo funkciju `rekurzija` koja kao ulazne podatke prima varijablu `duljina_podataka`, koja označava broj elemenata tablice, i dva niza tipa `Rijec` te vraća popunjenu tablicu.

```
Fixpoint rekurzija (duljina_podataka:nat) (r1 r2:Rijec ):=
  match duljina_podataka with
    | 0 => {|podaci := nil; broj_stupaca := length r2;|}
    | S m => iter (rekurzija m r1 r2) r1 r2
  end.
```

Konačno, dolazimo do ranije definirane funkcije `vratiRijec`. Funkcija `vratiRijec` radi backtracking po tablici. Pri implementaciji backtrackinga, ukoliko su elementi nizova koje uspoređujemo različiti, krećemo se u smjeru većeg susjednog prethodnika u tablici, a (radi određenosti) prema gore ako su jednaki.

Kako bismo funkciju učinili lakšom za korištenje i izbjegli ručno računanje ulaznih podataka koji su nam potrebni za funkciju `vratiRijec`, napisali smo funkciju `najdulji_zajednicki_podniz` koja vraća željeni podniz. Koristimo ranije spomenuti konstrukt `let ... in` u kojemo privremeno imenujemo tablicu sa `gen_tablica`. Funkcija `rekurzija` je definirana rekurzivno te se započinje pozivom na zadnjem elementu tablice — zbog čega je potrebno unaprijed znati veličinu tablice, tj. duljinu liste `podaci`. Duljinu liste `podaci` ćemo izračunati koristeći Coqove funkcije `Nat.mul` i `length`. Funkcija `vratiRijec` backtracking započinje na posljednjem elementu tablice pa njegovu poziciju računamo pomoću Coqovih funkcija `pred` (prethodnik) i `length`.

```
Definition najdulji_zajednicki_podniz (r1 r2: Rijec) : Rijec :=
  let gen_tablica := rekurzija (mul (length r1) (length r2)) r1 r2
  in vratiRijec gen_tablica r1 r2 (pred (length r1))
                                  (pred (length r2)).
```

Za kraj, pozovimo funkciju `najdulji_zajednicki_podniz` na prethodno navedenim nizovima:

```
Compute najdulji_zajednicki_podniz
           (1::1::2::2::4::4::3::3::nil)
           (1::2::1::2::4::3::4::3::1::nil).
```

Rezultat ispisa:

```
= 1 :: 1 :: 2 :: 4 :: 4 :: 3 :: nil
    : Rijec
```

Poglavlje 2

Račun induktivnih konstrukcija

2.1 Uvodno o računu induktivnih konstrukcija

Formalizam koji se nalazi u pozadini interaktivnog asistenta za konstrukciju dokaza, Coqa, nazivamo **Račun induktivnih konstrukcija** (*The Calculus of Inductive Constructions*, skraćeno CIC). To je jezik koji istovremeno nudi mogućnost pisanja programa u funkcijskom stilu, kao i mogućnost pisanja dokaza koristeći logiku višeg reda. CIC ima mogućnost prikaza mnogih struktura podataka, od uobičajenih tipova poput listā ili binarnih stabala pa sve do logičkih izjava, funkcija višeg reda i univerzumā. *Induktivne definicije* predstavljaju prirodnu reprezentaciju koncepata poput dostupnosti ili operacijske semantike definirane korištenjem pravila zaključivanja. Induktivne definicije, kao primitivni ili kao izvedeni koncept, jedan su od glavnih sastojaka jezika za interaktivno dokazivanje i reprezentaciju objekata i logičkih koncepata. Za dodatno proširiti raspravu o temi, pogledati [3].

2.2 Račun konstrukcija

Temelj za CIC je **Račun konstrukcija** (*The Calculus of Constructions*), koji možemo shvatiti kao uniformni sustav tipova („*pure type system*", skraćeno PTS). PTS je tipizirani λ-račun s jedinstvenom sintaksom za terme i tipove. Termi uključuju varijable, tipizirane apstrakcije (anonimne funkcije) i njihove pozive kao u uobičajenom λ-računu. Pri tome apstrakcije pišemo **fun** $x : A \Rightarrow t$, a pozive $t\ u$. Tip apstrakcije je (zavisni) produkt $\prod_{x:A} B$ što omogućava da je tip B izraza t ovisan o vrijednosti varijable x. Notacija $A \to B$ (koja označava tip funkcija koje idu s tipa A u tip B) je samo skraćenica u posebnom slučaju kada B ne ovisi o varijabli x. Tipovi također imaju tipove, koje zovemo **sortama**. Postoji

barem jedna sorta imena **Type**. PTS-ovi se razlikuju po sortama zadanima na početku i konstrukcijama koje možemo provoditi s tipovima. Uzmimo za primjer tip A; tada je funkcija identiteta **fun** $x\colon A \Rightarrow x$ term tipa $A \to A$. Ako A označava proizvoljni tip, možemo konstruirati polimorfinu identitetu **fun** $A\colon Type \Rightarrow$ **fun** $x\colon A \Rightarrow x$ tipa $\prod_{A\colon Type} (A \to A)$.

U Računu konstrukcija imamo beskonačni skup sorti $S := \{\mathsf{Prop}, \mathsf{Type}_1, \mathsf{Type}_2, \ldots\}$. Sorta Prop obuhvaća izraze koji označavaju logičke tvrdnje. Pratimo Curry–Howardovu korespondenciju gdje je tvrdnja A predstavljena tipom (uglavnom tipom dokaza za A), a dokaz tvrdnje A će odgovarati objektu t tipa A. S druge strane, tip t možemo shvatiti kao tvrdnju „tip t je neprazan", pa će svaki dobro oblikovani term tipa t poslužiti kao dokaz za tu tvrdnju. Ako A i B odgovaraju tvrdnjama, tada kondicional $A \to B$ predstavljamo tipom funkcija koje preslikavaju dokaze od A u dokaze od B. Tvrdnja $A \wedge B$ će biti predstavljena tipom $A \times B$ parova dokaza za A i dokaza za B. Ako uzmemo tip T, $\prod_{x\colon T} B$ će označavati tip zavisnih funkcija koje će za dani $t\colon T$ računati term tipa $B[t/x]$, a taj tip odgovara dokazima logičkih tvrdnji $(\forall x \in T)\, B(x)$. Tako tipovi predstavljaju logičke izjave pa će jezik sadržavati i prazne tipove, koji odgovaraju lažnim izjavama.

Napomena (oznake): Koristit ćemo notaciju $\forall x\colon T, B$ umjesto $\prod_{x\colon A} B$ kad je B izjava. Pišemo $t[u/x]$ za izjavu t u kojoj je varijabla x zamijenjena termom u. Term $t\,u_1 \ldots u_n$ predstavlja $(\ldots(t\,u_1)\ldots u_n)$ i **fun** $(x_1\colon A_1)\ldots(x_n\colon A_n) \Rightarrow t$ (notacija koju koristimo umjesto $\prod_{(x_1\colon A_1)\ldots(x_n\colon A_n)} B$) jednaka je **fun** $x_1\colon A_1 \Rightarrow \ldots$ **fun** $x_n\colon A_n \Rightarrow t$ (predstavlja $\prod_{(x_1\colon A_1)} \ldots \prod_{(x_n\colon A_n)} B$). Term $A \to B \to C$ grupiramo kao $A \to (B \to C)$, a $\prod_{x\colon A} B \to C$ kao $\prod_{x\colon A} (B \to C)$. Ponekad izostavljamo tip varijable iz apstrakcija i produkata kada je jasan iz konteksta. U Coqovoj logici višeg reda, tvrdnje i objekti se pišu korištenjem istog funkcijskog jezika s apstrakcijama i aplikacijama. Recimo, binarna relacija na tipu A će biti $R : A \to A \to Prop$. Možemo izgraditi izjavnu funkciju **fun** $x\colon A \Rightarrow R\,x\,x$ koja predstavlja skup objekata koji su u relaciji sami sa sobom tipa $A \to Prop$. Ovaj izraz ne smijemo pomiješati s izrazom kojim opisujemo refleksivnost, $\forall x\colon A, R\,x\,x$ (tipa $Prop$).

Želimo da je $A \to A \to Prop$ ispravno oblikovan tip, što znači da će on biti neke sorte. Ova sorta se zove $Type_1$. Ako želimo primijeniti istu konstrukciju na $Type_1$, ispravna oblikovanost znači da moramo uvesti novu sortu $Type_2$ koja će predstavljati tip objekta $Type_1$. Potreba za beskonačnom hijerarhijom se pojavljuje jer su jednostavniji PTS-ovi koji imaju samo jednu sortu $Type$ tipa $Type$ nekonzinstentni (Girardov paradoks).

2.3 Sudovi u CIC-u

CIC rukuje *sudovima* oblika:

$$x_1 : A_1, \ldots, x_n : A_n \vdash t : A. \tag{2.1}$$

U ovom sudu, $x_1 : A_1, \ldots, x_n : A_n$ nazivamo **kontekst**, a $t : A$ s desne strane znaka $\vdash$ nazivamo **zaključak**. Deklarirali smo da je varijabla x_i tipa A_i i ona predstavlja ime objekta tog tipa. Kada A_i označava logičku propoziciju, x_i je ime pretpostavke da A_i vrijedi. Sud možemo pročitati kao: pod pretpostavkom da su nam dani objekti x_i tipa A_i, term t označava objekt tipa A.

Pokušajmo na ovaj način opisati prirodne brojeve u Peanovom smislu. Uvodimo tip $N : Type$ koji predstavlja prirodne brojeve, objekt $z : N$ kojim označavamo nulu i objekt $S : N \to N$ za funkciju sljedbenik. S Γ_S ćemo označiti konekst $N : Type,\ z : N,\ S : N \to N$. Možemo izvesti sljedeće sudove:

$$\Gamma_S \vdash z : N, \qquad \Gamma_S \vdash S\,z : N, \qquad \Gamma_S \vdash S\,(S\,z) : N. \tag{2.2}$$

Sada možemo uvesti binarnu relaciju *le* koja predstavlja standardni uređaj na N. Dodajemo $le : N \to N \to Prop$ i hipeteze $lez : \forall x : N,\ le\,z\,n$ i $leS : \forall x\,y : N,\ le\,x\,y \to le\,(S\,x)(S\,y)$. Definiramo novi kontekst Γ_N koji proširuje Γ_S sljedećim deklaracijama:

$$le : N \to N \to Prop, \tag{2.3}$$

$$lez : (\forall x : N,\ le\,z\,x), \tag{2.4}$$

$$leS : (\forall x\,y : N,\ le\,x\,y \to le\,(S\,x)(S\,y)). \tag{2.5}$$

Koristeći dane hipoteze, moći ćemo izvesti sljedeće sudove:

$$\Gamma_N \vdash lez\,z : le\,z\,z, \tag{2.6}$$

$$\Gamma_N \vdash leS\ z\,z\,(lez\,z) : le\,(S\,z)(S\,z). \tag{2.7}$$

Pomoću indukcije možemo za svaki prirodni broj n pronaći term l_n takav da $\Gamma_N \vdash l_n : le\,(S^n z)(S^n z)$. U logici prvog reda, princip indukcije za svojstvo P je različit za svaku instancu svojstva P reprezentiranog formulom pa imamo beskonačno aksioma, odnosno matematička indukcija je *shema* aksioma. U logici višeg reda poput CIC-a, dovoljno je dodati jedan aksiom koji će obuhvatiti sve instance, jer možemo kvantificirati po varijabli P. Uvodimo:

$$\Gamma_P = \Gamma_N,\ ind : (\forall P : N \to Prop,\ P\,z \to (\forall x : N,\ P\,x \to P\,(S\,x)) \to \forall x : N,\ P\,x). \tag{2.8}$$

Sada možemo izvesti sud:

$$\Gamma_P \vdash \mathit{ind}\,(\,\mathbf{fun}\, x : N \Rightarrow \mathit{le}\, x\, x\,)(\,\mathit{lez}\, z\,)(\,\mathbf{fun}\,(\,x : N\,)(\,I : \mathit{le}\, x\, x\,) \Rightarrow \mathit{leS}\, x\, x\, I\,) : (\forall x : N,\, \mathit{le}\, x\, x). \tag{2.9}$$

Provjera tipa dokaznih terma može biti kompliciranа i korisnici bi to obično htjeli izbjeći. Oni to često mogu učiniti tako da ih konstruiraju pomoću programa više razine zvanih **taktike**. Međutim, dokazni term će uvijek biti izgrađen u pozadini i predan na provjeru tipova *jezgri* koja će provjeriti da nema grešaka u dokazu. Dokazni term može biti provjeren neovisno u različitim programima, a informacije u njemu mogu se uporabiti za analizu ovisnosti i pametan ispis.

2.4 Pravila zaključivanja

Formalno, CIC-ov sud je oblika $\Gamma \vdash t : A$ gdje je Γ kontekst, a t i A su dva terma, pri čemu A označava tip. Sud bez zaključka $\Gamma \vdash$ označava samo da je kontekst Γ ispravno oblikovan.

Navodimo pravila zaključivanja koja odgovaraju funkcionalnom dijelu CIC-a pri čemu je $S \supseteq \{\mathit{Prop}, \mathit{Type}_1, \mathit{Type}_2, \dots\}$.

$$\frac{\Gamma \vdash}{\Gamma \vdash \mathit{Prop} : \mathit{Type}_1} \tag{2.10}$$

$$\frac{\Gamma \vdash}{\Gamma \vdash \mathit{Type}_i : \mathit{Type}_{i+1}} \tag{2.11}$$

$$\frac{\Gamma \vdash \qquad (x : A) \in \Gamma}{\Gamma \vdash x : A} \tag{2.12}$$

$$\frac{\Gamma \vdash A : s \qquad s \in S \qquad (x : A) \notin \Gamma}{\Gamma, x : A \vdash} \tag{2.13}$$

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \mathbf{fun}\, x : A \Rightarrow t : \prod\limits_{x:A} B} \tag{2.14}$$

$$\frac{\Gamma \vdash f : \prod\limits_{x:A} B \qquad \Gamma \vdash a : A}{\Gamma \vdash f\, a : B[a/x]} \tag{2.15}$$

$$\frac{\Gamma, x : A \vdash B : \mathit{Prop}}{\Gamma \vdash \prod\limits_{x:A} B : \mathit{Prop}} \tag{2.16}$$

$$\frac{\Gamma, x : A \vdash B : \mathit{Type}_i \qquad \Gamma \vdash A : \mathit{Type}_i}{\Gamma \vdash \prod\limits_{x:A} B : \mathit{Type}_i} \tag{2.17}$$

$$\frac{\Gamma \vdash t : A \qquad s \in S \qquad \Gamma \vdash B : s \qquad A \preccurlyeq B}{\Gamma \vdash t : B} \tag{2.18}$$

Komentirat ćemo neka navedena pravila. Postoje dva pravila, (2.16) i (2.17) za određivanje tipa produkta $\prod_{x:A} B$. U oba slučaja term B mora imati valjani tip ako imamo $x : A$, i njegov tip bi trebao biti sorta s. Za $s = Prop$, produkt ostaje u $Prop$ iako je A možda dio većeg *univerzuma*. Primjerice, $\prod_{X:Prop} X \to X$ je tipa $Prop$. Kažemo da je $Prop$ **nepredikativna** sorta jer možemo izgraditi nove objekte u $Prop$ koristeći univerzalnu kvantifikaciju po klasi svih objekata u $Prop$ (uključujući i onaj trenutno definiran). Nepredikativni PTS-ovi su moćni, ali i vrlo osjetljivi jer nepredikativnost u kombinaciji s ostalim značajkama može lako dovesti do nekonzistentnosti. Uzmimo kao primjer $Prop$ tipa $Type_1$; nepredikativnost od $Type_1$ daje nekonzinstentni sistem, pa ako želimo izgraditi tip $\prod_{X:Type_1} X \to X$ trebamo veći univerzum $Type_2$.

Također nam je zanimljivo pravilo (2.18), poznato i kao *pravilo konverzije*. Term tipa A možemo promatrati kao term tipa B ako je B ispravno oblikovan i vrijedi relacija $A \preccurlyeq B$. Ova relacija služi dvjema svrhama. Prvo, ona implementira kumulativnost univerzuma, $Prop \preccurlyeq Type_1, Type_1 \preccurlyeq Type_2, \ldots$: svaki tip jednog univerzuma može se smatrati tipom kasnijeg univerzuma. Osim toga, implementira činjenicu da se s tipovima može računati (pravilom β u λ-računu) i da se rezultat tog računa smatra konvertibilnim početnom izrazu. Primjerice, $(\mathbf{fun}\, x : Type_1 \Rightarrow x)\, Prop \preccurlyeq Prop$ (i obrnuto). Pravilo β implementira činjenicu da se apstrakcija terma t nad varijablom x primijenjena na term a ponaša kao term t u kojem je x zamijenjen s a: $(\mathbf{fun}\, x : A \Rightarrow t)\, a \backsimeq t[a/x]$.

Prikažimo jedan od zanimljivih načina na koji ovo možemo koristiti. Kvantifikacija $\forall C : Prop, C$ je logička tvrdnja (term tipa $Prop$) koja kodira laž ($\bot$): ne postoji zatvoreni term tipa $\forall C : Prop, C$ (pa ni dokaz za $\bot$ bez hipoteza) i oblik dokaza t od $\forall C : Prop, C$ kojim možemo izgraditi dokaz $t\, C$ za proizvoljnu tvrdnju C je prirodno pravilo dedukcije za eliminaciju $\bot$.

Moguće je konstruirati egzistencionalnu kvantifikaciju (pomoću negacie univerzalne kvantifikacije):

$$(\exists\, x : A,\ B) := (\forall C : Prop,\ (\forall x : A,\ B \to C) \to C). \tag{2.19}$$

U prirodnoj dedukciji su izvediva pravila uvođenja i eliminacije za egzistencionalnu kvantifikaciju:

$$\frac{\Gamma \vdash B\,|t/x|}{\Gamma \vdash \exists\, x : A, B}, \qquad \frac{\Gamma \vdash p : B|t/x|}{\Gamma \vdash \mathbf{fun}\, C(H : \forall x : A, B \to C) \Rightarrow H\ t\ p : (\exists x : A, B)}. \tag{2.20}$$

$$\frac{\Gamma \vdash \exists x : A, B \quad \Gamma, B \vdash C \quad x \notin \Gamma, C}{\Gamma \vdash C}, \quad \frac{\Gamma \vdash t : \exists x : A, B \quad \Gamma, x : A, p : B \vdash u : C \quad x \notin \Gamma, C}{\Gamma \vdash t\, C(\mathbf{fun}\, (x : A)(p : B) \Rightarrow u) : C}. \tag{2.21}$$

Moramo biti oprezni jer CIC implementira konstruktivnu logiku umjesto klasične. Zato snažnija forma eliminacije

$$\frac{\Gamma, \neg C \vdash \bot}{\Gamma \vdash C} \tag{2.22}$$

nije općenito dokaziva. Također možemo dokazati $\exists x, B \vdash \neg\forall x, \neg B$, ali ne i suprotni smjer. Egzistencijalna kvantifikacija u CIC-u je snažnija nego u klasičnoj logici u smislu da ćemo pri dokazu za $\exists x : A, B$ uvijek moći naći term t takav da je $B[t/x]$ dokazivo, dok u klasičnoj logici možemo dobiti postojanje konačnog broja terma $t_1, \ldots t_k$ takvih da je $B[t_1/x] \vee \cdots \vee B[t_k/x]$ dokazivo za egzistencijalne formule.

Kvantifikacija u logici višeg reda se može iskoristiti za predstavljanje logičkih relacija. Primjerice, *Leibnizovu jednakost* $x = y$ za x i y tipa A, možemo enkodirati koristeći tvrdnju $\forall P : A \to Prop, Px \to Py$. U CIC-u su izvediva dva pravila za njeno uvođenje i eliminaciju:

$$\frac{}{\Gamma \vdash t = t}, \qquad \frac{\Gamma \vdash t = u \quad \Gamma, x : A \vdash B : Prop \quad \Gamma \vdash B[t/x]}{\Gamma \vdash B[u/x]}. \tag{2.23}$$

2.5 Induktivno definirani tipovi

Induktivne definicije uvodimo kao nadogradnju čistog računa konstrukcija. Glavna zadaća induktivnih definicija je pružiti nam učinkovit prikaz tipova podataka. Nova induktivna defincija dodaje se tako da navedemo njeno ime, mjesnost (tip induktivne defincije) i skup njenih *konstruktora*.

Primjer: Prirodne brojeve definirane s konstruktorima z i S možemo uvesti koristeći sljedeću deklaraciju:

```
Inductive N : Type := z :  N | S : N → N.
```

Ovakav način deklaracije stvara nove tipizirane objekte $N : Type$, $z : N$ i $S : N \to N$. Za razliku od slučaja prvog reda, ovdje logika obuhvaća činjenicu da je N *inicijalna algebra* obzirom na dvije navedene operacije. Posljedično, koristeći incijalna svojstva moći ćemo izgraditi funkcije tipa $N \to A$ i dokazati tvrdnje poput $\forall x : N, S\ x \neq z$ ili
$\forall x\, y : N,\ S\ x = S\ y \to x = y$.

Induktivne definicije možemo koristiti i za definiranje relacija. Koristeći prije navedene aksiome možemo definirati uređaj na prirodnim brojevima:

```
Inductive le : N → N → Prop :=
      | lez : ∀ x, le z x
      | leS : ∀ x y, le x y → le (S x) (S y).
```

Ova deklaracija uvodi $le : N \to N \to Prop$, $lez : (\forall x : N\ ,\ le\, z\, x)$ i
$leS\ : (\forall x\, y : N\ ,\ le\, x\, y \to le(S\, x)(S\, y))$ u kontekst. Sad također znamo da je *le* najmanja relacija R koja zadovoljava $\forall x : N, R\, z\, x$ i $\forall x\, y : N, R\, x\, y \to R\, (S\, x)\, (S\, y)$. To nam daje snažan alat za dokazivanje lema oblika $\forall x\, y : N, le\, x\, y \to R\, x\, y$.

Induktivne definicije mogu biti parametrizirane. Uzmimo kao primjer refleksivno-tranzitivno zatvorenje (Kleenejevu zvijezdu) binarne relacije R na tipu A, koje možemo induktivno definirati kao induktivni tip *RT* s parametrima A i R:

```
Inductive RT A (R : A → A → Prop) :  A → A → Prop :=
     | RTrefl :  ∀ x, RT A R x x
     | RTR : ∀ x y, R x y → RT A R x y
     | RTtran :  ∀ x y z, RT A R x z → RT A R z y → RT A R x y.
```

U ovoj definiciji imamo tri konstruktora: *RTrefl* nam kaže da je refleksivno-tranzitivno zatvorenje refleksivno, *RTR* da sadrži R i *RTtran* da je tranzitivno. Primjer tvrdnje koja se sada može induktivno dokazati je refleksivno-tranzitivno zatvorenje relacije sljedbenik na N ekvivaletno prethodno definiranoj relaciji *le* (iako je taj dokaz izvan opsega ovog rada):

```
∀ x y, le x y ↔ RT N ( fun x y ⇒ y = S x ),
```

Induktivne definicije koristimo i kako bismo lakše prikazali logičke operacije kao što su neistina (definirana bez konstruktora), egzenstencijalni kvantifikator ili jednakost.

```
Inductive False :  Prop := .
Inductive ex A (P : A → Prop) : Prop := exists : ∀ x, P x → ex A P.
Inductive eq A ( x :  A ) : A → Prop := eqrefl :  eq x x.
```

U logici prvog reda, kako uvodimo aksiome da bismo oblikovali neku teoriju, postoji rizik da ona nema model, odnosno da je kontradiktorna. Ovo se ne može dogoditi s induktivnim definicijama: postoje sintaksna ograničenja na tipove konstruktora koja osiguravaju postojanje modela. Više o tome može se naći u [3].

Poglavlje 3

Taktike i dokazi

U prvom poglavlju smo se već susreli s dokazima (pri dokazivanju obligacija), a u ovom poglavlju ćemo ih detaljnije obraditi. Proći ćemo kroz jednostavnije primjere dokaza kako bismo se upoznali s osnovnim taktikama u Coqu. Više o tim taktikama, kao i popis svih taktika u Coqovoj standardnoj biblioteci, može se vidjeti u [1].

3.1 Primjer: asocijativnost disjunkcije

Definiramo propoziciju `A` kojom iskazujemo asocijativnost disjunkcije.

```
Proposition A: forall P Q R: Prop, P \/ (Q \/ R) <-> (P \/ Q) \/ R.
```

Coq nam vraća koji cilj moramo dokazati: tvrdnju propozicije `A`. Nakon njenog dokaza, propoziciju `A` moći ćemo koristiti (npr. pomoću taktike `apply`) u dokazima drugih tvrdnji.

```
1 goal
______________________________________(1/1)
forall P Q R : Prop, (P \/ (Q \/ R)) <-> ((P \/ Q) \/ R)
```

Početak dokaza označavamo ključnom riječju `Proof`. Nakon nje koristimo tatktike kako bismo dokazali zadane ciljeve.

```
Proof.
```

Prva taktika koju koristimo je `intros`. Njome započinjemo dokaz tako da pretpostavimo da imamo neke `P`, `Q` i `R` tipa `Prop` te smo time obradili univerzalnu kvantifikaciju s početka iskaza cilja. To je korak koji u neformalnim matematičkim dokazima obično provodimo rečenicom poput „Neka su P, Q i R proizvoljne formule”.

```
intros P Q R.
```

```
1 goal
P, Q, R : Prop
______________________________________(1/1)
(P \/ (Q \/ R)) <-> ((P \/ Q) \/ R)
```

Bikondicional $C \leftrightarrow D$ je pokrata za konjunkciju dva kondicionala $(C \rightarrow D) \wedge (D \rightarrow C)$ pa zapravo moramo dokazati konjunkciju. Taktika koju koristimo kako bismo konjunkciju razdvojili na dva konjunkta je `split`. Primjenom taktike `split` dobili smo dva cilja koja je potrebno dokazati, a svaki sadrži po jedan kondicional.

```
split.
```

```
2 goals
P, Q, R : Prop
______________________________________(1/2)
(P \/ (Q \/ R)) -> (P \/ Q) \/ R
______________________________________(2/2)
((P \/ Q) \/ R) -> P \/ (Q \/ R)
```

Ranije smo se upoznali sa meta-taktikom crtica koju koristimo kako bismo se koncentrirali na prvi cilj.

```
1 goal
P, Q, R : Prop
______________________________________(1/1)
(P \/ (Q \/ R)) -> (P \/ Q) \/ R
```

Ponovno koristimo taktiku `intros` da bismo pretpostavili da vrijedi prvi dio implikacije te ga imenujemo sa `pretp`.

```
- intros pretp.
```

```
1 goal
P, Q, R : Prop
pretp : P \/ (Q \/ R)
______________________________________(1/1)
(P \/ Q) \/ R
```

U sljedećem koraku dokaza koristimo hipotezu `pretp`, tako što ćemo taktikom `destruct H as [ H1 | H2 ]` rastaviti dokaz na slučajeve. S `prvi` smo označili prvi član disjunkcije `(P)`, dok `drugi` označava drugi član disjunkcije `(Q ∨ R)`.

```
destruct pretp as [prvi|drugi].
```

```
2 goals
P, Q, R : Prop
prvi : P
______________________________________(1/2)
(P \/ Q) \/ R
______________________________________(2/2)
(P \/ Q) \/ R
```

Napomenimo da nam već taktika `intros` omogućava rastav na slučajeve pri uvođenju formula u kontekst, tako da smo mogli odmah na početku dokaza ovog cilja napisati `intros [prvi|drugi]`.

Ponovno imamo dva cilja, a meta-taktike nam omogućuju da se koncentriramo na određenu granu dokaza. Ovisno o razini indentacije redom koristimo meta-taktike `-`, `+` i `*`. Budući da se nalazimo na drugoj razini indentacije, korisitmo meta-taktiku plus. Njenim korištenjem ulazimo u slučaj gdje koristimo hipotezu `prvi`, koja kaže da vrijedi `P`.

```
1 goal
P, Q, R : Prop
prvi : P
______________________________________(1/1)
(P \/ Q) \/ R
```

Da bismo dokazali disjunkciju, dovoljno je dokazati lijevi (*left*) ili desni (*right*) disjunkt. Uviđamo da se `P` pojavljuje u prvom disjunktu cilja, pa se odlučimo koncentirati na lijevu stranu izraza koristeći taktiku `left`.

```
+ left.
```

```
1 goal
P, Q, R : Prop
prvi : P
______________________________________(1/1)
P \/ Q
```

Ponovno se P pojavljuje u prvom članu preostale disjunkcije pa još jednom koristimo istu taktiku.

```
    left.
```

```
1 goal
P, Q, R : Prop
prvi : P
______________________________________(1/1)
P
```

Vidimo da smo došli u kontekst u kojem se cilj nalazi među pretpostavkama, pa ga možemo razriješiti taktikom `assumption` koja upravo to govori. Time završavamo ovu granu dokaza te nam Coq ispisuje preostale ciljeve: to su druga grana dokaza i drugi kondicional.

```
    assumption.
```

```
This subproof is complete, but there are some unfocused goals:

______________________________________(1/2)
(P \/ Q) \/ R
______________________________________(2/2)
(P \/ Q) \/ R -> P \/ Q \/ R
```

Ostajući na istoj (drugoj) razini ugniježdenosti, koristimo meta-taktiku plus kako bi ušli u drugu granu dokaza.

```
1 goal
P, Q, R : Prop
drugi : Q \/ R
______________________________________(1/1)
(P \/ Q) \/ R
```

Uočavamo da hipoteza `drugi` u sebi sadrži disjunkciju pa ponovno korisitmo taktiku `destruct`. Ponovno se dokaz grana u dvije grane.

```
    + destruct drugi as [ pretpQ | pretpR ].
```

```
2 goals
P, Q, R : Prop
pretpQ : Q
______________________________________(1/2)
(P \/ Q) \/ R
______________________________________(2/2)
(P \/ Q) \/ R
```

Kako se već nalazimo na drugoj razini ugniježdenosti, za ulazak u novu podgranu (na dubini 3) koristimo meta-taktiku zvjezdice.

```
1 goal
P, Q, R : Prop
pretpQ : Q
______________________________________(1/1)
(P \/ Q) \/ R
```

Korisitmo hipotezu `pretpQ` te uviđamo da prvi član disjunkcije cilja sadrži `Q` pa korisitmo taktiku `left.`

```
    * left.
```

```
1 goal
P, Q, R : Prop
pretpQ : Q
______________________________________(1/1)
P \/ Q
```

Sada je `Q` s desne strane pa koristimo taktiku `right` za dokazivanje drugog disjunkta.

```
    right.
```

```
1 goal
P, Q, R : Prop
pretpQ : Q
______________________________________(1/1)
Q
```

Uviđamo da se trenutni cilj nalazi među pretpostavkama pa korisitmo taktiku `assumption`. Time smo završili ovu podgranu dokaza te nam Coq ispisuje preostale ciljeve koje trebamo dokazati.

```
    assumption.
```

```
This subproof is complete, but there are some unfocused goals:

_____________________________________(1/2)
(P \/ Q) \/ R
_____________________________________(2/2)
((P \/ Q) \/ R) -> P \/ (Q \/ R)
```

Ponovno koristimo meta-taktiku zvjezdica kako bi se usredotočili na prvi navedeni cilj.

```
1 goal
P, Q, R : Prop
pretpR : R
_____________________________________(1/1)
(P \/ Q) \/ R
```

Uviđamo je da je `pretpR` jednaka desnoj strani cilja pa nakon nje koristimo taktiku `right`.

```
    * right.
```

```
1 goal
P, Q, R : Prop
pretpR : R
_____________________________________(1/1)
R
```

Ponovno dolazimo do jednakosti cilja i jedne od pretpostavki, pa taktikom `assumption` završavamo ovu stranu dokaza.

```
    assumption.
```

```
This subproof is complete, but there are some unfocused goals:

_____________________________________(1/1)
((P \/ Q) \/ R) -> P \/ (Q \/ R)
```

Preostalo nam je dokazati drugi kondicional čiji dokaz je analogan dokazu prvog kondicionala. Zbog toga navodimo samo korake u dokazu bez povratnih poruka. Kako bi čitatelj lakše pratio, navodimo cijeli dokaz.

```
Proposition A: forall P Q R: Prop, P \/ (Q \/ R) <-> (P \/ Q) \/ R.
Proof.
  intros P Q R. split.
  - intros pretp.
    destruct pretp as [prvi|drugi].
    + left. left. assumption.
    + destruct drugi as [ pretpQ | pretpR ].
      ++ left. right. assumption.
      ++ right. assumption.
  - intros pretp2.
    destruct pretp2 as [prvi|drugi].
    +  destruct prvi as [ pretpP | pretpQ].
       ++ left. assumption.
       ++ right. left. assumption.
    +  right. right. assumption.
Qed.
```

3.2 Prikaz osnovnih taktika i njihovih primjena

Nakon što smo prošli kroz osnovne taktike, a kako bismo si olakšali buduće konstruiranje dokaza, u tablicu ćemo popisati najčešće korištene taktike ovisno o danom cilju ili hipotezi. U tablici se koncentriramo na taktike usmjerene na logičke veznike.

	Hipoteza H	cilj
$\Rightarrow$	`apply H`	`intros H`
$\forall$	`apply H, specialize (H t)`	`intros x`
$\exists$	`destruct H as (x & H1)`	`exists t`
$\wedge$	`destruct H as (H1 & H2)`	`split`
$\vee$	`destruct H as [H1 | H2], case H`	`left, right`
$\neg$	`apply H, contradict, contradiction`	`intros, contra`
$=$	`rewrite H, rewrite <- H, subst`	`reflexivity, ring, lia`
`False`	`elim H`	`contradiction`

Ilustriramo korištenje tablice na primjeru. Kako bismo iskoristili hipotezu `H` u slučaju kada ona u sebi sadrži logički veznik $\vee$, upotrijebiti ćemo taktiku `destruct H as [H1 | H2]` pomoću koje ćemo razgranati dokaz na slučajeve ovisno o hipotezi.

```
1 goal
P, Q, R : Prop
H : P \/ (Q \/ R)
______________________________________(1/1)
(P \/ Q) \/ R
```

Kao rezultat dobivamo sljedeći izlaz.

```
2 goals
P, Q, R : Prop
H1 : P
______________________________________(1/2)
(P \/ Q) \/ R
______________________________________(2/2)
(P \/ Q) \/ R
```

Razrješavamo slučaj uz pretpostavku `H1` kao što je prezentirano u potpoglavlju 3.1, a nakon toga nam preostaje dokazati dani cilj uz pretpostavku `H2`.

```
1 goal
P, Q, R : Prop
H2 : Q \/ R
______________________________________(1/1)
(P \/ Q) \/ R
```

Ovaj slučaj je također razriješen u potpoglavlju 3.1.

3.3 Primjer: koraci u dokazu da $\sqrt{2} \notin \mathbb{Q}$

Cilj nam je dokazati da ne postoje dva relativno prosta prirodna broja koja su takva da je kvadrat prvog jednak dvostrukom kvadratu drugog broja. Zapisano:

```
¬ ∃ m n: nat (relpr m n ∧ kv n = 2 * kv m).
```
(3.1)

Definiramo funkciju kvadrat broja `n` kao:

```
Definition kv n := n * n.
```

Kako bismo definirali relaciju `relpr` kojom želimo odrediti jesu li dva broja relativno prosta, prvo nam je potrebna funkcija `dijeli`.

```
Definition dijeli m n := exists k, n = m * k.
```

Definiramo relaciju `relpr`:

```
Definition relpr m n :=
    forall d, dijeli d m -> dijeli d n -> d = 1.
```

Kako bismo dokazali tvrdnju (3.1), potrebno je prvo dokazati tvrdnju koju zovemo `lema1`:

```
(∀ n: nat) (paran n ∨ neparan n).
```
(3.2)

Zapišimo tu lemu u Coqu.

```
Lemma lema1: forall n, paran n \/ neparan n.
```

Funkcije paran i neparan definirane su na sljedeći način:

```
Definition paran := dijeli 2.
Definition neparan n := exists m, paran m /\ n = S m.
```

Uočimo da funkcija `dijeli` po definiciji prima dva argumenta, od kojih smo jedan unaprijed zadali u definiciji funkcije `paran`. Ekvivalentan zapis definicije `paran` bio bi:

```
Definition paran n := dijeli 2 n.
```

Ovdje koristimo *η-ekspanziju*: λ `x.f x = f`. Primijenjeno na funkciju `paran`:
`dijeli = λ m . λ n . dijeli m n ,`
`dijeli 2 = λ n. dijeli 2 n = λ n . paran n = paran.`

Dokaz tvrdnje `lema1`

Sada možemo dokazati tvrdnju (3.2).

```
Proof.
  intros n.
  induction n.
  - left. unfold paran. unfold dijeli. exists 0. lia.
  - destruct IHn as [H1 | H2].
    + right. unfold neparan. exists n.
      split.
      * assumption.
      * reflexivity.
    + left. unfold paran.
      unfold neparan in H2.
      destruct H2 as (m & n_neparan).
```

```
    destruct n_neparan as (m_paran & n_sljedenik).
    unfold paran in m_paran.
    unfold dijeli.
    unfold dijeli in m_paran.
    destruct m_paran as (k & m_2_k).
    exists (S k). subst n. subst m.
    lia.
Qed.
```

Proći ćemo kroz dijelove dokaza u kojima koristimo taktike s kojima se još nismo susreli. Dokaz započinjemo meta-naredbom `Proof`.

```
Proof.
```

```
1 goal
______________________________________(1/1)
forall n : nat, paran n \/ neparan n
```

Koristimo taktiku `intros` s kojom smo se susreli u potpoglavlju 3.1.

```
intros n.
```

```
1 goal
n : nat
______________________________________(1/1)
paran n \/ neparan n
```

Dokazat ćemo da tvrdnja vrijedi indukcijom po `n`, za što koristimo taktiku `induction`. Potrebno je dokazati da tvrdnja vrijedi za `n = 0` (baza indukcije) i za `S n`, uz pretpostavku da vrijedi za `n` (korak indukcije).

```
induction n.
```

```
2 goals
______________________________________(1/2)
paran 0 \/ neparan 0
______________________________________(2/2)
paran (S n) \/ neparan (S n)
```

Proći ćemo kroz neke taktike korištene u koraku indukcije. `IHn` je pretpostavka indukcije (kratica `IHn` dolazi od *Induction Hypothesis on* `n`), a potrebno je dokazati sljedeće:

```
1 goal
n : nat
IHn : paran n \/ neparan n
______________________________________(1/1)
paran (S n) \/ neparan (S n)
```

Koristimo taktiku `destruct H as [H1 | H2]` s kojom smo se upoznali u potpoglavlju 3.1 i dobivamo dva slučaja za koje trebamo dokazati zadani cilj.

```
destruct IHn as [H1 | H2].
```

```
2 goals
n : nat
H1 : paran n
______________________________________(1/2)
paran (S n) \/ neparan (S n)
______________________________________(2/2)
paran (S n) \/ neparan (S n)
```

Usredotočimo se na prvi slučaj:

```
1 goal
n : nat
H1 : paran n
______________________________________(1/1)
paran (S n) \/ neparan (S n)
```

U hipotezi `H1` imamo pretpostavku da je `n` paran, pa će `S n` biti neparan i zato koristimo taktiku `right`.

```
+ right.
```

```
1 goal
n : nat
H1 : paran n
______________________________________(1/1)
neparan (S n)
```

Znamo da je funkcija `neparan` u svojoj definiciji sadrži funkciju `paran` te kako bi iskoristili pretpostavku da je `n` paran, koristimo taktiku `unfold`. Taktikom `unfold` ćemo zamijeniti zadani izraz njegovom definicijom.

```
unfold neparan.
```

```
1 goal
n : nat
H1 : paran n
______________________________________(1/1)
exists m : nat, paran m /\ S n = S m
```

Potrebno je pronaći postoji li `m` koji je paran i čiji sljedbenik je jednak sljedbeniku od `n`. Vidimo da `n` zadovoljava navedene uvjete pa koristimo taktiku `exists` kojom označavmo da smo pronašli objekt koji zadovoljava tražene uvjete.

```
exists n.
```

```
1 goal
n : nat
H1 : paran n
______________________________________(1/1)
paran n /\ S n = S n
```

Pronađeni `m` je jednak `n` te je `n` uvršten umjesto `m`. U cilju imamo konjunkciju pa koristimo taktiku `split` opisanu u potpoglavlju 3.1.

```
split.
```

```
2 goals
n : nat
H1 : paran n
______________________________________(1/2)
paran n
______________________________________(2/2)
S n = S n
```

Usredotočimo se na dokaz prvog cilja.

```
1 goal
n : nat
H1 : paran n
______________________________________(1/1)
paran n
```

Budući da se cilj nalazi među hipotezama, dovoljno je upotrijebiti taktiku `assumption`.

```
    * assumption.
```

```
This subproof is complete, but there are some unfocused goals:

______________________________________(1/2)
S n = S n
______________________________________(2/2)
paran (S n) \/ neparan (S n)
```

Da bismo dokazali sve iz konjunkcije, preostalo nam je dokazati:

```
1 goal
n : nat
H1 : paran n
______________________________________(1/1)
S n = S n
```

Budući da su izrazi identični s obje strane jednakosti, koristimo taktiku `reflexivity`.

```
    * reflexivity.
```

```
This subproof is complete, but there are some unfocused goals:

______________________________________(1/1)
paran (S n) \/ neparan (S n)
```

Kako bismo dokazali korak indukcije, preostaje nam dokazati cilj i kada nam je zadana hipoteza `H2`.

```
1 goal
n : nat
H2 : neparan n
______________________________________(1/1)
paran (S n) \/ neparan (S n)
```

Uz hipotezu `H2` da je `n` neparan, treba nam lijeva strana cilja te koristimo taktiku `left`. Raspisat ćemo definicije `paran` i `neparan` pomoću taktike `unfold`. Kada taktiku `unfold` koristimo na hipotezi, potrebno je navesti njeno ime.

```
    left. unfold paran. unfold neparan in H2.
```

```
1 goal
n : nat
H2 : exists m : nat, paran m /\ n = S m
______________________________________(1/1)
dijeli 2 (S n)
```

Kada nam se egzistencijalni kvantifikator pojavi u cilju, koristimo taktiku `exists`, a kada je u hipotezi taktiku `destruct H as (x & H1)`.

```
destruct H2 as (m & n_neparan).
```

```
1 goal
n, m : nat
n_neparan : paran m /\ n = S m
______________________________________(1/1)
dijeli 2 (S n)
```

Preskačemo dio dokaza koji koristi sve do sad navedene taktike i dolazimo do ispisa:

```
1 goal
n, m, k : nat
m_2_k : m = 2 * k
n_sljedbenik : n = S m
______________________________________(1/1)
S n = 2 * S k
```

Među hipotezama pronalazimo hipotezu `n_sljedbenik` u kojoj imamo `n` definiran preko `m`. Upotrebljavamo taktiku `subst` kako bismo u cilju zamijenili `n` koristeći jednakost iskazanu u hipotezi `n_sljedbenik`.

```
subst n.
```

```
1 goal
m, k : nat
m_2_k : m = 2 * k
______________________________________(1/1)
S (S m) = 2 * S k
```

Na isti način koristimo taktiku `subst` kako bismo `m` zamijenili s njegovom vrijednošću iz hipoteze `m_2_k`.

```
    subst m.
```

```
1 goal
k : nat
______________________________________(1/1)
S (S (2 * k)) = 2 * S k
```

Preostali cilj jednostavno možemo razriješiti koristeći taktiku `lia`. Taktika `lia` je dio biblioteke `Lia`, a koristimo ju kao alat za aritmetičke manipulacije izrazima.

```
    lia.
```

```
No more goals.
```

Time je dokaz leme gotov.

```
    Qed.
```

Dokaz leme `lema2`

Sljedeća tvrdnja koju je potrebno dokazati je:

$$(\forall\ \texttt{n: nat})\ \neg(\texttt{paran n} \wedge \texttt{neparan n}). \tag{3.3}$$

Zapišimo tu lemu u Coqu.

```
    Lemma lema2:  forall n, ~ (paran n /\ neparan n).
```

Imamo sve potrebne alate potrebne za dokaz tvrdnje (3.3) pa samo navodimo njen dokaz.

```
Proof.
  intros n.
  intros pretp.
  destruct pretp as (n_paran & n_neparan).
  unfold neparan in n_neparan.
  destruct n_neparan as (m & H2).
  destruct H2 as (m_paran & n_sljedebnik).
  subst n.
  unfold paran in n_paran.
  unfold paran in m_paran.
  unfold dijeli in m_paran.
```

```
  destruct m_paran as (k & m_dvostruki).
  unfold dijeli in n_paran.
  subst m.
  destruct n_paran as (k0 & jednakost).
  lia.
Qed.
```

Dokaz leme `lema3`

Sljedeća lema glasi:

$$(\forall\ n\colon \mathrm{nat})\ (\mathrm{paran}\ n \leftrightarrow \mathrm{paran}\ (\mathrm{kv}\ n)). \tag{3.4}$$

Zapišimo je u Coqu.

```
Lemma lema3: forall n, paran n <-> paran (kv n).
```

Navodimo dokaz tvrdnje (3.4).

```
Proof.
  intros n.
  split.
  - intros n_paran.
    unfold paran in n_paran. unfold dijeli in n_paran.
    destruct n_paran as (k & n_dvostruki).
    subst n. exists (2 * kv k).
    unfold kv. lia.
  - intros pretp.
    unfold paran in pretp. unfold dijeli in pretp.
    destruct pretp as (k & kvadrat_n).
    assert (paran n \/ neparan n) as [n_paran|n_neparan] by apply lema1.
    + assumption.
    + exfalso.
      unfold neparan in n_neparan.
      destruct n_neparan as (m & H).
      destruct H as (m_paran & n_sljedbenik).
      subst n. unfold paran in m_paran. unfold dijeli in m_paran.
      destruct m_paran as (k0 & m_dvostruki).
      subst m.
      unfold kv in kvadrat_n.
      lia.
Qed.
```

U dokazu tvrdnje 3.4 smo koristili taktike koje do sad nisu objašnjenje pa ćemo detaljnije proći kroz neke dijelove dokaza. Lakši smjer za dokazati je da ako je n paran, tada je i kvadrat od n paran. Prikazat ćemo dokaz suprotnog smjera: ako je kvadrat od n paran, tada je i n paran.

```
1 goal
n : nat
______________________________________(1/1)
paran (kv n) -> paran n
```

Koristimo taktike s kojima smo se već upoznali u ovom poglavlju.

```
intros pretp.
unfold paran in pretp. unfold dijeli in pretp.
destruct pretp as (k & kvadrat_n).
```

```
1 goal
n, k : nat
kvadrat_n : kv n = 2 * k
______________________________________(1/1)
paran n
```

Koristimo taktiku `assert` koja nam omogućuje da u dokaz dodamo nove pretpostavke. Koristimo lemu `lema1` koja nam kaže da n mora biti paran ili neparan te time dobivamo dva slučaja. U prvom slučaju moramo dokazati dosadašnji cilj uz dodatnu pretpostavku da je n paran. Nova pretpostavka je imena `n_paran`. Analogno za drugi slučaj.

```
assert (paran n \/ neparan n) as [n_paran|n_neparan]
 by apply lema1.
```

```
2 goals
n, k : nat
kvadrat_n : kv n = 2 * k
n_paran : paran n
______________________________________(1/2)
paran n
______________________________________(2/2)
paran n
```

Prvi slučaj lako razriješimo jer je cilj sadržan u hipotezi `n_paran`.

```
    + assumption.
```

```
This subproof is complete, but there are some unfocused goals:

_____________________________________(1/1)
paran n
```

Sada je potrebno dokazati drugi slučaj gdje je pretpostavka da je n neparan.

```
1 goal
n, k : nat
kvadrat_n : kv n = 2 * k
n_neparan : neparan n
_____________________________________(1/1)
paran n
```

Cilj nam je dokazati da ovaj slučaj vodi u kontradikciju jer ne možemo dokazati da je n paran uz pretpostavku da je n neparan, a ako n nije paran, tada ni kvadrat od n ne može biti paran. Koristimo taktiku `exfalso` kako bismo cilj zamijenili s `False`.

```
    + exfalso.
```

```
1 goal
n, k : nat
kvadrat_n : kv n = 2 * k
n_neparan : neparan n
_____________________________________(1/1)
False
```

U daljnjim koracima dokaza koristimo taktike s kojima smo se već ranije upoznali u ovome poglavlju pa ostatak dokaza nećemo detaljno prolaziti.

Dokaz leme `lema4`

Kako bismo si olakšali dokaz tvrdnje (3.1), uvodimo lemu:

$$\texttt{(∀ u v: nat) 2 * u = 4 * v → u = 2 * v.} \tag{3.5}$$

Zapišimo je u Coqu.

```
    Lemma lema4: forall u v, 2 * u = 4 * v -> u = 2 * v.
```

Dokaz tvrdnje (3.5) je sasvim aritmetički pa je dovoljno iskoristiti taktiku `lia`.

```
Proof.
    lia.
Qed.
```

Dokaz leme `lema5`

Prije dokaza tvrdnje teorema preostaje nam još uvesti lemu koja kaže da parni brojevi sigurno nisu relativno prosti.

```
(∀ m n: nat) (paran m ∧ paran n → ¬ relpr m n).
```
(3.6)

Kako bismo si olakšali dokaz u Coqu, odnosno primjenu leme, zapisat ćemo je na drugačiji način:

```
(∀ m n: nat) (paran m → paran n → ¬ relpr m n).
```
(3.7)

Zapišimo je sada u Coqu.

```
Lemma lema5: forall m n, paran m -> paran n -> ~ relpr m n.
```

Dokaz tvrdnje (3.7):

```
Proof.
  intros m n paran_m paran_n suprotno.
  unfold relpr in suprotno. specialize (suprotno 2).
  unfold paran in *.
  apply suprotno in paran_m.
  - discriminate.
  - assumption.
Qed.
```

U dokazu koristimo taktike s kojima se još nismo susreli, a budući da je dokaz kratak, možemo proći kroz cijeli dokaz.

```
1 goal
_____________________________________(1/1)
forall m n : nat, paran m -> paran n -> ~ relpr m n
```

Vidimo da je potrebno uvesti hipoteze pomoću taktike `intros`, pri čemu na kraju pretpostavljamo suprotno, tj. uvodimo hipotezu koja kaže da su `m` i `n` relativno prosti. Uobičajeno smo to radili ovako:

```
intros m n.
intros paran_m.
intros paran_n.
intros suprotno.
```

Možemo skratiti zapis stavljajući sve hipoteze zajedno i dobit ćemo jednak rezultat.

```
intros m n paran_m paran_n suprotno.
```

```
1 goal
m, n : nat
paran_m : paran m
paran_n : paran n
suprotno : relpr m n
______________________________________(1/1)
False
```

Koristimo taktiku `unfold` kako bismo raspisali definiciju od `relpr` u hipotezi `suprotno`.

```
unfold relpr in suprotno.
```

```
1 goal
m, n : nat
paran_m : paran m
paran_n : paran n
suprotno : forall d : nat, dijeli d m -> dijeli d n -> d = 1
______________________________________(1/1)
False
```

Kako bismo iskoristili pretpostavke da su `m` i `n` parni brojevi, želimo `d` u iskazu hipoteze `suprotno` postaviti na 2. To činimo pomoću taktike `specialize`.

```
specialize (suprotno 2).
```

```
1 goal
m, n : nat
paran_m : paran m
paran_n : paran n
suprotno : dijeli 2 m -> dijeli 2 n -> 2 = 1
______________________________________(1/1)
False
```

Kako bismo iskoristili pretpostavke da su `m` i `n` parni, potrebno je prvo raspisati definicije `paran`. Budući da ne želimo posebno pisati naredbu `unfold` za svaku pretpostavku, možemo koristiti znak * koji će zadanu taktiku primijeniti na svako pojavljivanje definicije `paran` u ovom koraku dokaza (u kontekstu i u cilju) .

```
unfold paran in *.
```

```
1 goal
m, n : nat
paran_m : dijeli 2 m
paran_n : dijeli 2 n
suprotno : dijeli 2 m -> dijeli 2 n -> 2 = 1
______________________________________(1/1)
False
```

Koristimo taktiku `apply` kako bismo hipotezu `suprotno` iskoristili na hipotezi `paran_m`.

```
apply suprotno in paran_m.
```

```
2 goals
m, n : nat
paran_m : 2 = 1
paran_n : dijeli 2 n
suprotno : dijeli 2 m -> dijeli 2 n -> 2 = 1
______________________________________(1/2)
False
______________________________________(2/2)
dijeli 2 n
```

Uz pretpostavku da je `m` paran, ako vrijedi da je i `n` paran, iz iskaza hipoteze `suprotno` slijedi da je 2 = 1. U tom slučaju trebamo još dokazati cilj koji je jednak `False`.

```
1 goal
m, n : nat
paran_m : 2 = 1
paran_n : dijeli 2 n
suprotno : dijeli 2 m -> dijeli 2 n -> 2 = 1
______________________________________(1/1)
False
```

Hipoteza `paran_m` je nakon primjene hipoteze `suprotno` jednaka $2 = 1$ što je kontradikcija. Sada možemo upotrijebiti taktiku `discriminate` koja odbacuje ovaj slučaj jer smo se susreli s jednakošću koja je sama po sebi kontradiktorna. Još jedan primjer kada možemo upotrijebiti taktiku `discriminate` je u slučaju da imamo hipotezu koja sadrži `true = false`. U oba slučaja radi se o različitim konstruktorima induktivnog tipa (`nat` odnosno `bool`).

```
- discriminate.
```

```
This subproof is complete, but there are some unfocused goals:

_____________________________________(1/1)
dijeli 2 n
```

Preostaje nam još dokazati da je `n` paran.

```
1 goal
m, n : nat
paran_m : dijeli 2 m
paran_n : dijeli 2 n
suprotno : dijeli 2 m -> dijeli 2 n -> 2 = 1
_____________________________________(1/1)
dijeli 2 n
```

Budući da u pretpostavkama imamo hipotezu `paran_n`, dovoljno je iskoristiti `assumption`.

```
- assumption.
```

```
No more goals.
```

Time smo završili dokaz.

```
Qed.
```

Dokaz glavne tvrdnje: $\sqrt{2} \notin \mathbb{Q}$

Dokazali smo sve leme koje će nam biti od pomoći u dokazu tvrdnje (3.1) koju sada zapisujemo u obliku teorema:

```
Theorem sqrt2nQ: ~ exists m n, relpr m n /\ 2 * kv m = kv n.
```

Dokaz teorema navodimo niže.

```
Proof.
  intros (m & n & h1 & h2).
  assert (paran (kv n)) as h3.
  - unfold paran. unfold dijeli.
    exists (kv m).
    symmetry. assumption.
  - apply lema3 in h3.
    pose h3 as kopija_h3.
    unfold paran in kopija_h3.
    unfold dijeli in kopija_h3.
    destruct kopija_h3 as (k & dvostruki_n).
    rewrite dvostruki_n in h2.
    replace (kv (2*k)) with (4 * kv k) in h2.
    + apply lema4 in h2.
      assert (paran (kv m)) as h4.
      * unfold paran. unfold dijeli.
        exists (kv k). assumption.
      * apply lema3 in h4.
        apply (lema5 m n); assumption.
    + unfold kv. lia.
Qed.
```

Kako bismo opravdali potrebu dosadašnjih lema, ukratko prolazimo kroz dokaz. Potrebno je dokazati:

```
1 goal
______________________________________(1/1)
~ (exists m n : nat, relpr m n /\ 2 * kv m = kv n)
```

Inače bismo dokaz započeli:

```
intros suprotno.
destruct suprotno as (m & n & pretp1).
destruct pretp1 as (h1 & h2).
```

No možemo to kraće zapisati kao:

```
intros (m & n & h1 & h2).
```

```
1 goal
m, n : nat
h1 : relpr m n
h2 : 2 * kv m = kv n
______________________________________(1/1)
False
```

Ako pročitamo hipotezu h2, uviđamo da kvadrat od n mora biti paran jer je djeljiv s 2. Zbog toga dodajemo novu pretpostavku `paran (kv n)` imena h3 koristeći taktiku `assert`.

```
    assert (paran (kv n)) as h3.
```

```
2 goals
m, n : nat
h1 : relpr m n
h2 : 2 * kv m = kv n
______________________________________(1/2)
paran (kv n)
______________________________________(2/2)
False
```

Coq zahtjeva da dokažemo tvrdnju `paran (kv n)` prije nego što je uvrsti među hipoteze.

```
1 goal
m, n : nat
h1 : relpr m n
h2 : 2 * kv m = kv n
______________________________________(1/1)
paran (kv n)
```

Koristimo taktiku `unfold` kako bismo razmotali definicije `paran` i `dijeli` u cilju.

```
    - unfold paran. unfold dijeli.
```

```
1 goal
m, n : nat
h1 : relpr m n
h2 : 2 * kv m = kv n
______________________________________(1/1)
exists k : nat, kv n = 2 * k
```

Uviđamo da `kv m` zadovoljava tražene zahtjeve na `k` pa koristimo taktiku `exists`.

```
exists (kv m).
```

```
1 goal
m, n : nat
h1 : relpr m n
h2 : 2 * kv m = kv n
______________________________________(1/1)
kv n = 2 * kv m
```

Da bismo mogli iskoristi taktiku `assumption`, potrebno je da izrazi zamijene strane u cilju kako bi jednakost u potpunosti odgovarala hipotezi `h2`. Koristimo taktiku `symmetry` da bismo to učinili.

```
symmetry. assumption.
```

```
1 goal
m, n : nat
h1 : relpr m n
h2 : 2 * kv m = kv n
______________________________________(1/1)
2 * kv m = kv n
```

Sada uviđamo da su cilj i hipoteza `h2` jednaki te koristimo taktiku `assumption`. Time smo dokazali da je kvadrat od `n` paran te je hipoteza `h3` dodana u pretpostavke.

```
1 goal
m, n : nat
h1 : relpr m n
h2 : 2 * kv m = kv n
h3 : paran (kv n)
______________________________________(1/1)
False
```

Sada možemo iskoristiti lemu `lema3` koja tvrdi da je kvadrat od `n` paran ako i samo ako je `n` paran. Time hipoteza `h3` sada tvrdi da je `n` paran.

```
apply lema3 in h3.
```

```
1 goal
m, n : nat
h1 : relpr m n
h2 : 2 * kv m = kv n
h3 : paran n
______________________________________(1/1)
False
```

Kako bismo mogli raspisati tvrdnju da je n paran, a da pritom ne izgubimo tvrdnju u prvotnom obliku, koristimo taktiku pose koja stvara kopiju hipoteze h3.

```
pose h3 as kopija_h3.
```

```
1 goal
m, n : nat
h1 : relpr m n
h2 : 2 * kv m = kv n
h3 : paran n
kopija_h3 := h3 : paran n
______________________________________(1/1)
False
```

Sada možemo raspisati tvrdnju da je n paran, čime dobivamo n izražen preko k.

```
unfold paran in kopija_h3.
unfold dijeli in kopija_h3.
destruct kopija_h3 as (k & dvostruki_n).
```

```
1 goal
m, n : nat
h1 : relpr m n
h2 : 2 * kv m = kv n
h3 : paran n
k : nat
dvostruki_n : n = 2 * k
______________________________________(1/1)
False
```

Koristeći taktiku rewrite možemo iskoristiti hipotezu dvostruki_n kako bismo n u hipotezi h2 zamijenili s 2 * k. Želimo dokazati da i m mora biti paran.

```
    rewrite dvostruki_n in h2.
```

```
1 goal
m, n : nat
h1 : relpr m n
k : nat
h2 : 2 * kv m = kv (2 * k)
h3 : paran n
dvostruki_n : n = 2 * k
______________________________________(1/1)
False
```

Da bismo mogli iskoristiti ranije dokazanu lemu, moramo 2 iz izraza `2*k` izbaciti izvan kvadrata i zato koristimo taktiku `replace`. Taktika `replace` nam omogućava da dio izraza zamijenimo ekvivalentnim podizrazom, no i dodaje novi cilj u kojem zahtjeva da dokažemo su ti podizrazi ekvivalentni.

```
    replace (kv (2 * k)) with (4 * kv k) in h2.
```

```
2 goals
m, n : nat
h1 : relpr m n
k : nat
h2 : 2 * kv m = 4 * kv k
h3 : paran n
dvostruki_n : n = 2 * k
______________________________________(1/2)
False
______________________________________(2/2)
4 * kv k = kv (2 * k)
```

Prvo dokazujemo dosadašnji cilj.

```
1 goal
m, n : nat
h1 : relpr m n
k : nat
h2 : 2 * kv m = 4 * kv k
h3 : paran n
dvostruki_n : n = 2 * k
```

```
_____________________________________(1/1)
False
```

Koristimo lemu `lema4` 3.5 kako bismo skratili izraz u hipotezi `h2` s 2.

```
    apply lema4 in h2.
```

```
1 goal
m, n : nat
h1 : relpr m n
k : nat
h2 : kv m = 2 * kv k
h3 : paran n
dvostruki_n : n = 2 * k
_____________________________________(1/1)
False
```

Jednako kao što smo dokazali da je `n` paran, dokazujemo da je kvadrat od `m` paran.

```
    assert (paran (kv m)) as h4.
      *  unfold paran. unfold dijeli.
         exists (kv k). assumption.
```

```
1 goal
m, n : nat
h1 : relpr m n
k : nat
h2 : kv m = 2 * kv k
h3 : paran n
dvostruki_n : n = 2 * k
h4 : paran (kv m)
_____________________________________(1/1)
False
```

Ponovno koristimo lemu `lema3` kako bismo iz pretpostavke da je kvadrat od `m` paran dobili da je `m` paran.

```
    * apply lema3 in h4.
```

```
1 goal
m, n : nat
h1 : relpr m n
k : nat
h2 : kv m = 2 * kv k
h3 : paran n
dvostruki_n : n = 2 * k
h4 : paran m
______________________________________(1/1)
False
```

Među pretpostavkama sada imamo hipoteze da su `m` i `n` parni te da su relativno prosti. Sada možemo iskoristiti lemu `lema5` koristeći taktiku `apply` te naznačimo da su zadani `m` i `n`.

```
apply (lema5 m n).
```

```
3 goals
m, n : nat
h1 : relpr m n
k : nat
h2 : kv m = 2 * kv k
h3 : paran n
dvostruki_n : n = 2 * k
h4 : paran m
______________________________________(1/3)
paran m
______________________________________(2/3)
paran n
______________________________________(3/3)
relpr m n
```

Potrebno je dokazati navedene ciljeve, no pošto su svi oni sadržani među hipotezama možemo iskoristiti meta-taktiku `;` kako bismo na sve dobivene ciljeve primijenili taktiku `assumption`.

```
apply (lema5 m n); assumption.
```

```
This subproof is complete, but there are some unfocused goals:
```

```
_____________________________________(1/1)
4 * kv k = kv (2 * k)
```

Preostaje nam dokazati da jednakost koju smo ranije iskoristili. Koristimo taktiku `unfold` kako bismo dobili čisto aritmetički izraz bez naših (korisničkih) definicija.

```
  + unfold kv.
```

```
1 goal
m, n : nat
h1 : relpr m n
k : nat
h2 : 2 * kv m = kv (2 * k)
h3 : paran n
dvostruki_n : n = 2 * k
_____________________________________(1/1)
4 * (k * k) = 2 * k * (2 * k)
```

Sada možemo iskoristiti taktiku `lia` i njome završiti dokaz.

```
  lia. Qed.
```

```
No more goals.
```

Bibliografija

[1] https://coq.inria.fr/refman/proof-engine/tactics.html.

[2] Bertot, Yves: *Coq in a Hurry*. HAL open science, 2015.

[3] Paulin-Mohring, Christine: *Introduction to the Calculus of Inductive Constructions*. All about Proofs, Proofs for All, (55), 2015.

Sažetak

U prvom poglavlju upoznajemo Coq kao funkcijski programski jezik. Poglavlje završavamo primjerom rješavanja problema nalaženja najduljeg zajedničkog podniza neka dva niza u Coqu.

Drugo poglavlje posvećujemo Računu induktivnih konstrukcija (CIC). Opisujemo formalizam koji stoji u pozadini Coqa.

U trećem poglavlju navodimo osnovne smjernice pri konstruiranju dokaza, s fokusom na logičke veznike i kvantifikatore. Kroz primjere asocijativnosti disjunkcije i nekih koraka u dokazu iracionalnosti broja $\sqrt{2}$, upoznajemo se s najčešće korištenim taktikama.

Summary

In first chapter we are introduced to Coq as a functional programming language. The chapter ends with an example where we show how to solve the problem of finding the longest common subsequence of two given sequences using Coq.

We devoted the second chapter to The Calculus of Inductive Constructions (CIC). Here we are describing the formalism upon which Coq is based on.

In the third chapter, we are providing the basic guidlines to constructing proofs with focus on the logical connectives and quantifiers. We are getting to know the most often used tactics through the examples of associativity of disjunction and some of the steps in the proof that $\sqrt{2}$ is irrational number.

Životopis

Rođena sam 28. 6. 1997. u Koprivnici. Završila sam Prirodoslovno-matematičku gimnaziju u Gimnaziji „Fran Galović" Koprivnica. 2016. godine sam upisala Prirodoslovno-matematički fakultet Sveučilišta u Zagrebu, preddiplomski studij Matematika, koji sam završila 2020. godine. Iste godine upisala sam diplomski studij na Prirodoslovno-matematičkom fakultetu, smjer Računarstvo.