Ovo je zahvala.

Sadržaj

1. Uvod

2. Coq

U ovom poglavlju dajemo pogled svisoka na programski sustav Coq. Prvo ćemo objasniti što je uopće Coq, u kojem je kontekstu nastao, i od kojih komponenti se sastoji. Zatim ćemo dati kratak pregled programiranja u Coqu, nakon čega ćemo se baviti naprednijim konceptima i spomenuti neka ograničenja. Za širi opseg gradiva, čitatelja upućujemo na knjige *Coq'Art* [1], *Software Foundations* [2, 3, 4] i *Certified Programming with Dependent Types* [5] te na službenu dokumentaciju [6].

2.1. Što je Coq?

Alat za dokazivanje Coq[1], punog naziva *The Coq Proof Assistant*, programski je sustav pomoću kojeg korisnici mogu dokazivati matematičke tvrdnje. **Misao:** *ne sluzi samo tome, moze biti i opceniti funkcijski programski jezik, moze sluziti za programiranje sa zavisnim tipovima* Alat se temelji na λ-računu i teoriji tipova, a prva je inačica implementirana godine 1984. [6] Ovaj rad koristi inačicu 8.18 iz rujna godine 2023.

Program Coq može se pokrenuti u interaktivnom ili u skupnom načinu rada. Interaktivni način rada pokreće se naredbom `coqtop`, a korisniku omogućuje rad u ljusci sličnoj `bash` i `python` ljuskama. Interaktivna ljuska (također poznata pod imenom *toplevel*) služi unosu definicija i iskazivanju lema. Skupni način rada pokreće se naredbom `coqc`, a korisniku omogućuje semantičku provjeru i prevođenje izvornih datoteka u jednostavnije formate. Kod formaliziranja i dokazivanja, korisnik će najčešće koristiti interaktivni način rada, po mogućnosti kroz neku od dostupnih razvojnih okolina. [2]

Misao: *spomenuti proof mode? spomenuti workflow dokazivanja? ima jedna zgodna*

[1] `https://coq.inria.fr/`

[2] Autor rada koristio je paket *Proof General* za uređivač teksta *Emacs*. Druge često korištene okoline su *VsCoq* i *CoqIDE*.

slika u QED at Large **Misao:** *treba naglasiti da je Coq interaktivni dokazivač teorema*

Kao programski jezik, Coq se sastoji od više podjezika različitih namjena, od kojih spominjemo *Vernacular*, *Gallinu* i *Ltac*.

Vernacular **Misao:** *vernacular znači „govorni jezik”* je jezik naredbi kojima korisnik komunicira sa sustavom (i u interaktivnom i u skupnom načinu rada); svaka Coq skripta (datoteka s nastavkom `.v`) je niz naredbi. Neke od najčešće korištenih naredbi su `Check`, `Definition`, `Inductive`, `Fixpoint` i `Lemma`. Pomoću naredbi za tvrdnje, kao što je `Lemma`, Coq prelazi iz *toplevela* **Misao:** *treba bolji prevod* u način dokazivanja (*proof mode*).

Gallina je Coqov strogo statički tipiziran specifikacijski jezik. Kako se glavnina programiranja u Coqu svodi upravo na programiranje u Gallini, posvećujemo joj idući odjeljak.

Ltac je Coqov netipiziran jezik za definiciju i korištenje taktika. Taktike su pomoćne naredbe kojima se u načinu dokazivanja konstruira dokaz. Može se reći da je Ltac jezik za metaprogramiranje Galline. Primjeri taktika su `intros`, `destruct`, `apply` i `rewrite`.

Pogledajmo ilustrativan primjer.

```
1  Lemma example_lemma : 1 + 1 = 2.
2  Proof.
3    cbn. reflexivity.
4  Qed.
```

Ključne riječi `Lemma`, `Proof` i `Qed` dio su Vernaculara, izraz `example_lemma : 1 + 1 = 2` dio je Galline, a pomoćne naredbe `cbn` i `reflexivity` dio su Ltaca.

Jezgra programskog sustava Coq je algoritam za provjeru tipova (*type checking*) implementiran u OCamlu — svaka tvrdnja koja se dokazuje izrečena je pomoću tipova. Ostatak sustava u načelu služi za knjigovodstvo i poboljšanje korisničkog iskustva. Nužno je da jezgra sustava bude relativno mala kako bismo se mogli uvjeriti u njenu točnost. U suprotnom, možemo li biti sigurni da su naše dokazane tvrdnje doista istinite? **Misao:** *kažem istinite, ali u stvari mislim dokazive, no to je nespretno za napisati i diskusija oko toga je preopćenita*

Prve inačice Coqa implementirale su račun konstrukcija, no kasnije je dodana podr-

ška za induktivno i koinduktivno definirane tipove [7, 8]. Danas se može reći da Coq implementira polimorfni kumulativni račun induktivnih konstrukcija [9]. **Misao:** *mogu spomenuti i preteče računa konstrukcija i jezike koji ih implementiraju, npr. Lisp je implementacija λ-računa* Coq se, osim kao dokazivač teorema, može koristiti i za programiranje sa zavisnim tipovima. U toj sferi konkuriraju jezici Agda[3], Idris[4] i Lean[5]. Coq se između njih ističe po usmjerenosti prema dokazivanju, posebno po korištenju taktika (jezik Ltac) i nepredikativnoj sorti `Prop` (o kojoj će kasnije biti riječi). Još jedna prednost Coqa je mehanizam *ekstrakcije* pomoću kojeg korisnik može proizvoljnu funkciju[6] prevesti u jezik niže razine apstrakcije.[7] Mehanizam ekstrakcije nije dokazano točan, no poželjno je da funkcije zadržavaju točnost i nakon ekstrakcije pa se radi na verifikaciji ekstrakcije [9].

2.2. Programiranje u Coqu

Gallina je funkcijski programski jezik, što znači da su funkcije prvoklasni objekti — funkcije mogu biti argumenti i povratne vrijednosti drugih funkcija. Dodatno, varijable su nepromjenjive (*immutable*) te se iteracija ostvaruje rekurzijom. Za uvod u funkcijsko programiranje, čitatelja upućujemo na knjigu *Programming in Haskell* [10]. Primjeri koje ćemo vidjeti u ostatku ovog odjeljka dijelom se oslanjanju na tipove i funkcije definirane u Coqovoj standardnoj knjižnici.[8]

Gallina je strogo statički tipiziran jezik, što znači da se svakom termu prilikom prevođenja dodjeljuje tip[9]. Naredbom `Check` možemo provjeriti tip nekog terma ili doznati da se termu ne može dodijeliti tip. Dalje u radu pod „term” mislimo na dobro formirane terme, odnosno na one terme kojima se može dodijeliti tip. Kažemo da je term *stanovnik* tipa koji mu je dodijeljen. Za tip kažemo da je *nastanjen*, odnosno *nenastanjen*, ako postoji, odnosno ne postoji, stanovnik tog tipa. Kako su u Coqu i tipovi termi, radi razumljivosti i zvučnosti umjesto „tip tipa” kažemo „sorta tipa”.

[3] `https://wiki.portal.chalmers.se/agda/`
[4] `https://www.idris-lang.org/`
[5] `https://lean-lang.org/`
[6] Za koju je dokazao točnost, štogod to značilo.
[7] Trenutno su podržani Haskell, OCaml i Scheme.
[8] `https://coq.inria.fr/library/`
[9] Tipovi su kolekcije srodnih objekata.

Kao i u ostalim jezicima, kod programiranja u Coqu korisnik se oslanja na dostupne primitivne izraze, od kojih su najvažniji:

- `forall`, pomoću kojeg se konstruiraju funkcijski tipovi i zavisni produkti;
- `match`, pomoću kojeg se provodi *pattern matching*;
- `fun`, pomoću kojeg se definiraju funkcije;
- `fix`, pomoću kojeg se definiraju rekurzivne funkcije i
- `cofix`, pomoću kojeg se definiraju korekurzivne funkcije.

Jedna od osnovnih naredbi za stvaranje novih terma je naredba `Definition`.

```
Definition negb (b : bool) : bool :=
match b with
| false => true
| true => false
end.
```

U gornjem kodu definirana je funkcija `negb` čiji se argument `b` tipa `bool` destrukturira te se vraća njegova negacija, također tipa `bool`. Važno je napomenuti da svaki `match` izraz mora imati po jednu granu za svaki konstruktor tipa.[10] U ovom su primjeru konstante `false` i `true` jedini konstruktori tipa *bool*. Funkcija `negb` je tipa `bool→bool`.

```
Definition mult_zero_r : Prop := forall (n : nat), n * 0 = 0.
```

Ovdje je definirana propozicija (tip) imena `mult_zero_r` kao univerzalno kvantificirana tvrdnja po prirodnim brojevima.

Naredbom `Inductive` definira se *induktivni* tip te se automatski za njega generiraju principi *indukcije* i *rekurzije*.

```
Inductive nat : Set :=
| O : nat
| S : nat -> nat.
```

Ovim kodom definirali smo tri terma:

- `nat` (tip prirodnih brojeva) je term sorte `Set`,
- `O` (broj nula) je term tipa `nat` i
- `S` (funkcija sljedbenika) je term tipa `nat→nat`.

Za term `nat` kažemo da je konstruktor tipa (*type constructor*), a za terme `O` i `S` kažemo da su konstruktori objekata (*object constructors*).

[10] U tandemu s uvjetom strukturalne rekurzije, ovime je osigurana totalnost svake funkcije.

Rekurzija nad induktivnim tipovima može se ostvariti naredbom `Fixpoint`, koja u pozadini koristi izraz `fix`.

```
Fixpoint plus (n m : nat) {struct n} : nat :=
match n with
| O => m
| S n' => S (plus n' m)
end.
```

U ovom primjeru definirana je funkcija `plus` koja prima dva argumenta tipa `nat`. Funkcija je rekurzivna po prvom argumentu što je vidljivo oznakom `{struct n}`. Napominjemo da su induktivni tipovi dobro utemeljeni, to jest svaki term induktivnog tipa je konačan.

Osim induktivnih, u Coqu postoje i koinduktivni tipovi, koji nisu dobro utemeljeni, zbog čega za njih nije moguće definirati principe indukcije i rekurzije. Umjesto rekurzije, koinduktivni tipovi koriste se u korekurzivnim funkcijama. Standardan primjer koinduktivnog tipa je beskonačna lista.

```
Set Primitive Projections.
CoInductive Stream (A : Type) := {
      hd : A;
      tl : Stream A;
    }.
```

Ovime smo definirali familiju tipova `Stream` indeksiranu tipskom varijablom `A`. Svaki `Stream` ima glavu i rep koji je također `Stream`.

Stanovnici koinduktivnih tipova konstruiraju se korekurzivnim funkcijama naredbom `CoFixpoint`, koja u pozadini koristi `cofix`.

```
CoFixpoint from (n : nat) : Stream nat := Cons _ n (from (n + 1)).
```

Ovime je definirana funkcija `from` koja za ulazni argument `n` vraća niz prirodnih brojeva od n na dalje.

Razlika induktivnih i koinduktivnih tipova može se izreći epigramom:

„Induktivni tipovi su domene rekurzivnih funkcija, a koinduktivni tipovi su kodomene korekurzivnih

Time se želi reći da se termi induktivnih tipova destrukturiraju u rekurzivnim funkci-

jama, dok se termi koinduktivnih tipova konstruiraju u korekurzivnim funkcijama.

Misao: *Barendregtova kocka, term koji ovisi o termu, term koji ovisi o tipu, tip koji ovisi o termu, tip koji ovisi o tipu*

2.3. Hijerarhija tipova

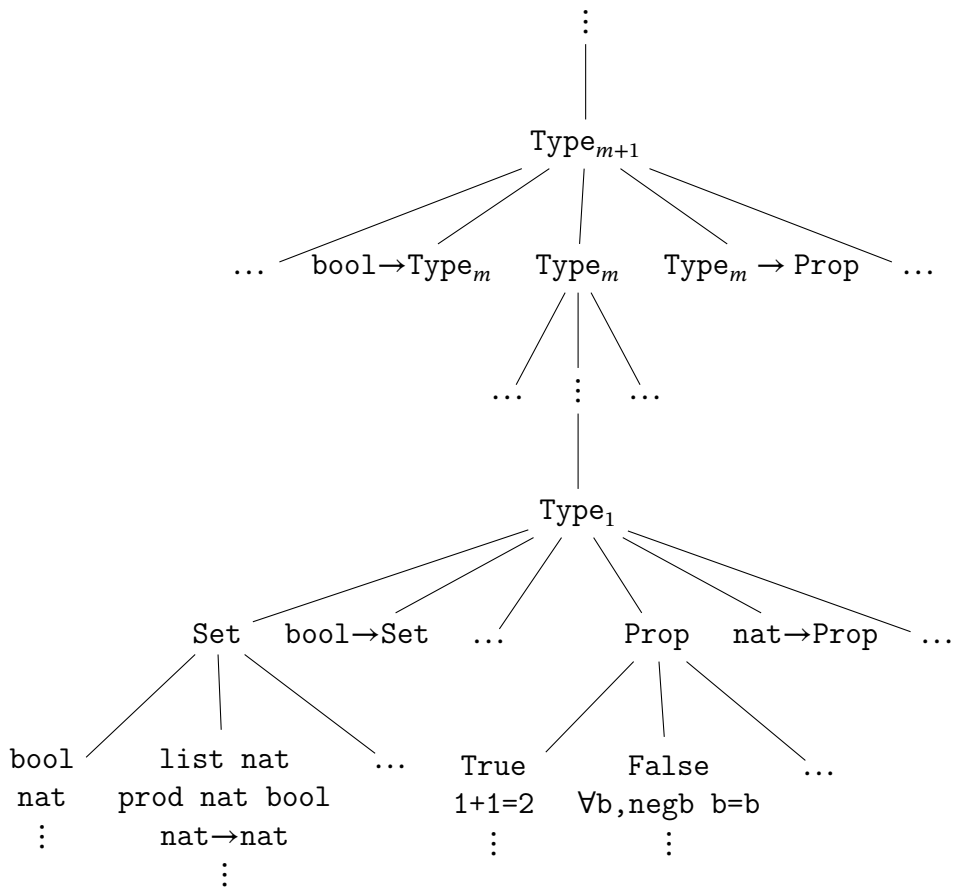
U usporedbi s tradicionalnim programskim jezicima, Coqov tipski sustav je ekspresivniji jer dopušta tipove koji mogu ovisiti o termima. Takvi tipovi se u Coqu konstruiraju `forall` izrazom. Primjer jednog takvog tipa je „lista duljine n", gdje je n neki prirodan broj, a čiji su stanovnici n-torke. Općeniti tip tada glasi „za svaki n, lista duljine n" — on ovisi o stanovniku drugog tipa (u ovom slučaju, tipa `nat`).

Spomenuli smo da su i tipovi termi te im se može dodijeliti sorta. Postoji li najveći tip, odnosno postoji li tip `Type` čiji su stanovnici svi dobro formirani termi? Prisjetimo se, svaki term ima svoj tip. Kada bi takav `Type` postojao, tada bi vrijedilo `Type : Type`, što može dovesti do paradoksa lašca[11]. Takav dokazivač teorema bio bi inkonzistentan te bismo njime mogli dokazati kontradikciju, čime dokazivač efektivno gubi svoju svrhu. Umjesto jedne „velike" sorte `Type`, u Coqu postoji niz monotono rastućih sorti $\texttt{Type}_n$ za sve prirodne brojeve n, takav da vrijedi $\texttt{Type}_n : \texttt{Type}_m$ kad god vrijedi $n < m$. Ilustracija ove **kumulativne hijerarhije tipova** prikazana je na slici 2.1. Dvije najvažnije sorte u Coqu su sorta `Set` i sorta `Prop`.

Naziv `Set` sinonim je za sortu $\texttt{Type}_0$, a njeni stanovnici su **mali tipovi**. **Misao:** *NB: Tu je prije pisalo da su stanovnici od `Set` programi, no to nije tako. Stanovnici sorte `Set` su mali tipovi (ne znam kako ih drugačije nazvati), a tek onda su stanovnici malih tipova programi.* Primjerice, tipovi `nat` i `bool` su mali tipovi. Dodatno, funkcije koje primaju i vraćaju male tipove su također mali tipovi. Tada su i produkti, sume, liste i stabla malih tipova također mali tipovi. Intuitivno se može reći da su mali oni tipovi s čijim se stanovnicima može efektivno računati. Stanovnike malih tipova nazivamo **programima**.

Stanovnici sorte `Prop` su **propozicije**. Za razliku od programa, sa propozicijama ne

[11] Ova rečenica je lažna. U teoriji skupova to je Russellov paradoks, u teoriji tipova to je Girardov paradoks



Slika 2.1. Kumulativna hijerarhija tipova

možemo efektivno računati, već ih moramo dokazivati. Stanovnici propozicija su njihovi **dokazi**.

Za dokazivače teorema, poželjna je mogućnost definicije predikata (propozicija) nad proizvoljnim tipovima. Zbog toga u Coqu prilikom definicije terma sorte `Prop` možemo raditi kvantifikaciju po proizvoljno velikim tipovima (što uključuje i sortu `Prop`).

```
1 Inductive isNat : Set -> Prop :=
2 | IsNat : isNat nat.
```

Tako je ovdje `isNat` predikat nad sortom `Set`, a u primjeru ispod `not` je predikat nad sortom `Prop`.

```
1 Definition not (P : Prop) := P -> False.
```

Ovaj stil kvantifikacije omogućuje nam definiciju proizvoljnih propozicija i propozicijskih veznika. Kažemo da je sorta `Prop` **nepredikativna**. S druge strane, sorta `Set` je **predikativna**, to jest *ne dopušta* kvantifikaciju po proizvoljno velikim tipovima. Na

praktičnoj strani, predikativnost ograničava korisnika da prilikom definicije programa smije koristiti samo druge programe.

2.4. Propozicije i tipovi, dokazi i termi

U decimalnom zapisu broja π, barem jedna znamenka pojavljuje se beskonačno mnogo puta. Doista, kada bi se svaka znamenka javljala samo konačno mnogo puta, broj π bio bi racionalan. Međutim, nije jasno *koja* znamenka ima to svojstvo. Da bismo odgovorili na to pitanje, morali bismo prebrojiti *sve* broja π, što nije moguće u konačno mnogo koraka.

Sličnim pitanjima bavili su se logičari dvadesetog stoljeća. *Klasični* logičari bi gornju tvrdnju smjesta prihvatili, dok bi *konstruktivisti* tražili konkretnu znamenku. Između ostalog, ovakva razmatranja rezultirala su fundamentalnim uvidom u povezanost programiranja i dokazivanja. Naime, želimo li dokazati konjunkciju, dovoljno je zasebno dokazati njene konjunkte. S druge strane, želimo li konstruirati par objekata, dovoljno je zapakirati prvi i drugi objekt u konstruktor para. Na sličan način, želimo li dokazati implikaciju, dovoljno je pretpostaviti njen antecedent te pomoću njega dokazati konzekvens. Ako pak želimo konstruirati funkciju, smijemo uzeti jedan argument i pomoću njega konstruirati povratnu vrijednost. Dodatno, nemoguće je dokazati laž, a istina trivijalno vrijedi. S druge strane, ako tip nema konstruktore, nije moguće definirati vrijednost tog tipa. Ako pak tip ima barem jedan konstruktor, tada postoje i njegovi stanovnici. Kroz ove primjere vidimo fenomen **Curry–Howardove korespondencije**, koju možemo sažeti epigramom:

„Propozicije su tipovi, dokazi su programi."

Time se dokazivanje svodi na programiranje. Pogledi na dvije strane ovog novčića mogu se vidjeti u tablici 2.1.

Za bolju ilustraciju, prikazujemo princip matematičke indukcije u Coqu. Prisjetimo se, za proizvoljni predikat P nad prirodnim brojevima, princip matematičke indukcije glasi:

$$P(0) \land (\forall n, P(n) \to P(n+1)) \to \forall n, P(n).$$

Dokazivanje	Programiranje
propozicija	tip
dokaz	program
laž	prazan tip
istina	nastanjen tip
konjunkcija	produktni tip
disjunkcija	zbrojni tip
implikacija	funkcijski tip
univerzalna kvantifikacija	zavisni produkt
egzistencijalna kvantifikacija	zavisni koprodukt

Tablica 2.1. Sličnosti dokazivanja i programiranja

Tvrdnju dokazujemo analizom broja n. Ako je $n = 0$, tvrdnja slijedi iz *baze* indukcije. Ako pak je $n = n' + 1$ za neki n', tada rekurzivno konstruiramo dokaz za $P(n')$, a konačna tvrdnja slijedi primjenom *koraka* indukcije na rekurzivno konstruirani dokaz.

```
Definition nat_ind (P : nat -> Prop)
                   (baza : P 0)
                   (korak : forall n, P n -> P (S n))
  : forall n, P n :=
  fix F (n : nat) : P n :=
    match n with
    | 0 => baza
    | S n' => korak n' (F n')
    end.
```

Za term `nat_ind` kažemo da je dokazni objekt (*proof object*) za tvrdnju matematičke indukcije. Princip matematičke indukcije je samo poseban slučaj **principa indukcije**, kojeg Coq automatski generira za svaki induktivno definiran tip.

2.5. Ograničenja tipskog sustava

Kao što smo već vidjeli, Coqov tipski sustav je ekspresivniji od tipskih sustava tradicionalnih programskih jezika. Međutim, kako bi se sačuvala poželjna svojstva algoritma provjere tipova, ipak se tipski sustav mora ograničiti.

Uvjet pozitivnosti odnosi se na definiciju induktivnih i koinduktivnih tipova. **Misao:** *NB: uvjet pozitivnosti mora vrijediti i u pozitivnim i u negativnim koinduktivnim ti-*

povima. Ovo ograničenje zabranjuje *negativne* pojave tipa kojeg definiramo u argumentima njegovih konstruktora.

```
Inductive Lam :=
| LamVar (n : nat)
| LamApp (M N : Lam)
| LamAbs (M : Lam -> Lam).
```

Pokretanje primjera iznad rezultira greškom `Non strictly positive occurrence of "Lam" in "(Lam -> Lam) -> Lam"` — drugim riječima, tip `Lam` se javlja negativno u konstruktoru `LamAbs`, odnosno kao argument funkcije koja je parametar konstruktora. Ovaj uvjet štiti korisnika od inkonzistentnosti, a za točnu definiciju pozitivnosti čitatelja upućujemo na dokumentaciju.[12] Uz uvjet pozitivnosti za induktivne tipove vezan je **uvjet strukturalne rekurzije**. Ovim uvjetom osigurava se totalnost rekurzivno definirane funkcije tako da se argument po kojem je funkcija rekurzivna strukturalno smanjuje u svakom koraku rekurzije.

Uvjet produktivnosti odnosi se na definiciju korekurzivnih funkcija, a dualan je uvjetu strukturalne rekurzije. Ovaj uvjet također štiti korisnika od inkonzistentnosti, a glasi: svaki korekurzivni poziv smije se javljati točno kao izravan argument konstruktora koinduktivnog tipa čiji element definiramo.[13] Zbog tog uvjeta, iduća definicija nije moguća.

```
Set Primitive Projections.
CoInductive NatStream := {
    nat_hd : nat;
    nat_tl : NatStream;
  }.

CoFixpoint foo : NatStream := foo.
```

Greška koju sustav javlja glasi `Unguarded recursive call in "foo"`, što znači da se korekurzivni poziv `foo` *ne* javlja kao izravni argument konstruktora. S druge strane, definicija

```
CoFixpoint bar : NatStream := {| nat_hd := 0; nat_tl := bar |}.
```

[12] https://coq.inria.fr/doc/v8.18/refman/language/core/inductive.html#well-formed-inductive-definitions

[13] https://coq.inria.fr/doc/v8.18/refman/language/core/coinductive.html#co-recursive-functions-cofix

je sasvim legalna. **Misao:** *Tu se baš i ne vidi zašto se* `bar` *javlja kao izravan argument konstruktora jer koristimo negativni koinduktivni tip. Ispod haube je to sve zamotano u neki* `Build_NatStream`.

Misao: *Bili smo spomenuli četiri ograničenja na sastanku. Uvjet strukturalne rekurzije i uvjet produktivnosti su na neki način dualni. Treće ograničenje bio je uvjet pozitivnosti. Čovjek očekuje da onda postoji nešto dualno uvjetu pozitivnosti za koinduktivne tipove. Nije li to opet uvjet pozitivnosti?*

Misao: *Ipak neću spomenuti razlike između pozitivnih i negativnih koinduktivnih tipova. Mislim da je to izvan dosega ovog rada.*

Posljednje ograničenje koje spominjemo vezano je uz irelevantnost dokaza (*proof irrelevance*). Naime, mnogi teoremi mogu se dokazati na više načina, ali svaki pojedini dokaz (dakle, postupak kojim smo od pretpostavka došli do konkluzije) *nije bitan*. Matematičarima su bitni samo iskaz teorema i činjenica da se teorem *može dokazati*. Sam postupak dokazivanja smatra se „implementacijskim detaljom”. Upravo zato analiza dokaza ima smisla samo kada se dokazuje, ali ne i kada se programira. U našoj terminologiji to znači da se *pattern matching* nad dokazima smije provoditi samo kod definiranja terma sorte `Prop`. U suprotnom, mogli bismo definirati programe koji ovise o *konkretnom dokazu*, umjesto o iskazanom teoremu. **Ograničenje eliminacije propozicije** nastalo je radi ekstrakcije — svi termi sorte `Prop` se „brišu” prilikom prevođenja iz Coqovog tipskog sustava u tipske sustave niže razine apstrakcije. Kada ovog ograničenja ne bi bilo, ekstrakcija u jednostavnije jezike naprosto ne bi bila moguća.

3. Logika prvog reda s induktivnim definicijama

U ovom poglavlju predstavljamo glavne rezultate diplomskog rada koji uključuju formalizaciju logike prvog reda s induktivnim definicijama FOL_{ID} te dokaznog sustava $LKID$, koje je prvi uveo Brotherston [11]. Definicije, leme i dokazi u ovom poglavlju preuzete su iz Brotherstonove disertacije [12].

Prvo ćemo definirati sintaksu i semantiku logike FOL_{ID}, nakon čega ćemo definirati njene standardne modele. Zatim ćemo prikazati dokazni sustav $LKID$ te konačno dokazati adekvatnost sustava $LKID$ s obzirom na standardnu semantiku, što je ujedno i glavni rezultat ovog diplomskog rada.

Svaka definicija i lema u ovom poglavlju bit će popraćena svojim pandanom u Coqu. Jedan je od ciljeva diplomskog rada prikazati primjene Coqa u matematici, zbog čega leme nećemo dokazivati „na papiru", već se dokaz svake leme može pronaći na GitHub repozitoriju rada.[1] Zainteresiranom čitatelju predlažemo interaktivni prolazak kroz dokaze lema.

Prije no što krenemo na formalizaciju, valja prokomentirati odnos matematičkog i Coqovog vokabulara. U matematici pojam „skup" može imati dva značenja; prvo se odnosi na skupove kao *domene diskursa*, dok se drugo odnosi na skupove kao *predikate*, odnosno podskupove. Primjerice, skup prirodnih brojeva $\mathbb{N}$ je domena diskursa kada je riječ o svim prirodnim brojevima te zbog toga pišemo $n \in \mathbb{N}$ umjesto $\mathbb{N}(n)$. S druge strane, skup svih parnih brojeva E je podskup skupa $\mathbb{N}$, a može se interpretirati kao predikat na prirodnim brojevima te pišemo $E(n)$ umjesto $n \in E$. U Coqu se skupovi kao domene diskursa formaliziraju tipovima sorte `Set`[2], dok se skupovi kao predikati forma-

[1] TODO: REPO LINK

[2] Ili općenito kao tipovi sorte `Type`.

liziraju funkcijama iz domene diskursa u sortu `Prop`. Na primjer, tip prirodnih brojeva `nat` je sorte `Set`, a predikat `Nat.Even` je tipa `nat→Prop`.

3.1. Sintaksa

Kao i u svakom izlaganju logike, na početku je potrebno definirati sintaksu.

Definicija 1 (Signatura). *Jezik prvog reda s induktivnim predikatima* (kratko signatura), u oznaci Σ, je skup simbola od kojih razlikujemo *funkcijske*, *obične predikatne* i *induktivne predikatne* simbole. Mjesnost simbola reprezentiramo funkcijom iz odgovarajućeg skupa simbola u skup $\mathbb{N}$. Funkcijski simboli mjesnosti nula nazivaju se *konstante*, a predikatni simboli mjesnosti nula nazivaju se *propozicije*.

```
Structure signature := {
  FuncS : Set;
  fun_ar : FuncS -> nat;
  PredS : Set;
  pred_ar : PredS -> nat;
  IndPredS : Set;
  indpred_ar : IndPredS -> nat
}.
```

Primjer 1. **Misao:** Σ_{PA}

U ostatku poglavlja promatramo jednu proizvoljnu, ali fiksiranu signaturu Σ. Fiksiranje nekog proizvoljnog objekta je česta pojava u matematici, prvenstveno zato što finksirane argumente ne trebamo spominjati eksplicitno. Coq omogućuje fiksiranje naredbom `Context`, pod uvjetom da se korisnik nalazi u `Section` okolini.[3] Većina definicija i lema u ovom radu su napisane upravo unutar `Section` okoline.

Definicija 2 (Term). *Varijabla* je prirodan broj. Skup terma konstruiramo rekurzivno na način:

1. svaka varijabla je term;
2. ako je f funkcijski simbol mjesnosti n te su $t_1, \ldots, t_n$ termi[4], onda je $f(t_1, \ldots, t_n)$ također term.

[3] `https://coq.inria.fr/doc/v8.18/refman/language/core/sections.html`

[4] Primijetimo, broj terma ovisi o mjesnosti funkcijskog simbola. U Coq implementaciji ovog „konstruktora" možemo vidjeti da je on zavisnog tipa.

```
Inductive term : Set :=
| var_term : var -> term
| TFunc : forall (f : FuncS Σ), vec term (fun_ar f) -> term.
```

Princip indukcije za `term` potrebno je ručno definirati. Naime, induktivni tip `term` je *ugniježđen* po konstruktoru `TFunc` što znači da se javlja omotan oko drugog induktivnog tipa[5] kao argument. Za ugniježđene induktivne tipove, Coq generira *neprikladne* principe indukcije.

```
Lemma term_ind
     : forall P : term Σ -> Prop,
         (forall v, P (var_term v)) ->
         (forall f args, (forall st, V.In st args -> P st) ->
                         P (TFunc f args)) ->
       forall t : term Σ, P t.
```

Definicija 3. Skup svih varijabli koje se javljaju u termu t, u oznaci $TV(t)$, konstruiramo rekurzivno na način:

1. za varijablu v vrijedi $TV(v) = \{v\}$,
2. za n-mjesni funkcijski simbol f i terme $t_1, \ldots, t_n$ vrijedi $TV(f(t_1, \ldots, t_n)) = \bigcup_{1 \leq i \leq n} TV(t_i)$.

```
Inductive TV : term -> var -> Prop :=
| TVVar : forall v, TV (var_term v) v
| TVFunc : forall f args v st, V.In st args ->
                               TV st v -> TV (TFunc f args) v.
```

Definicija 4 (Formula)**.** Skup formula konstruiramo rekurzivno na način:

1. ako je Q običan ili induktivan predikatni simbol mjesnosti n te su $t_1, \ldots, t_n$ termi, onda je $Q(t_1, \ldots, t_n)$ *atomarna* formula;
2. ako je φ formula, onda su $\neg\varphi$ i $\forall\varphi$ također formule;
3. ako su φ i ψ formule, onda je $\varphi \to \psi$ također formula.

```
Inductive formula : Set :=
| FPred (P : PredS Σ) : vec (term Σ) (pred_ar P) -> formula
| FIndPred (P : IndPredS Σ) : vec (term Σ) (indpred_ar P) -> formula
| FNeg : formula -> formula
| FImp : formula -> formula -> formula
| FAll : formula -> formula.
```

Za univerzalnu kvantifikaciju odstupamo od tradicionalne defincije formule. Umjesto

[5] Ovdje `vec`.

kvantificiranja po eksplicitnoj varijabli, mi ćemo implicitno kvantificirati po varijabli 0. Ovaj pristup kvantifikaciji[6], imena „de Bruijnovo indeksiranje", bitno olakšava rad sa supstitucijama, a uveden je u članku [13]. O samoj implementaciji de Bruijnovog indeksiranja više se može pročitati u knjizi *Types and Programming Languages* [14]. Za potrebe ovog rada koristili smo program *Autosubst2*[7] [15, 16] za automatsko generiranje tipova terma i formula te pripadajućih funkcija supstitucija i pomoćnih lema. **Misao:** *Ovaj odlomak preseliti nekamo na početak odjeljka.*

Definicija 5. Skup slobodnih varijabli formule φ, u oznaci $FV(\varphi)$, konstruiramo rekurzivno na način:

1. $FV(P(u_1, \ldots, u_n)) = \bigcup_{1 \leq i \leq n} TV(u_i)$,
2. $FV(\neg\varphi) = FV(\varphi)$,
3. $FV(\varphi \to \psi) = FV(\varphi) \cup FV(\psi)$,
4. $FV(\forall\varphi) = \{v \mid 1 + v \in FV(\varphi)\}$.

```
Inductive FV : formula -> var -> Prop :=
| FV_Pred : forall R args v st,
    V.In st args -> TV st v -> FV (FPred R args) v
| FV_IndPred : forall R args v st,
    V.In st args -> TV st v -> FV (FIndPred R args) v
| FV_Imp_l : forall F G v, FV F v -> FV (FImp F G) v
| FV_Imp_r : forall F G v, FV G v -> FV (FImp F G) v
| FV_Neg : forall F v, FV F v -> FV (FNeg F) v
| FV_All : forall F v, FV F (S v) -> FV (FAll F) v.
```

Definicija 6 (Supstitucija)**.** Supstitucija je svaka funkcija iz skupa $\mathbb{N}$ u skup terma, a domena joj se može rekurzivno proširiti na skup terma i skup formula.

```
Fixpoint subst_term  (σ : var -> term) (t : term) : term :=
  match t with
  | var_term v => σ v
  | TFunc f args => TFunc f (V.map (subst_term σ) args)
  end.
```

[6] Ili općenitije, vezivanju varijabli.
[7] https://github.com/uds-psl/autosubst2

```
Fixpoint subst_formula
  (σ : var -> term Σ) (φ : formula )
  : formula  :=
  match φ return formula  with
  | FPred P args => FPred P (V.map (subst_term σ) args)
  | FIndPred P args => FIndPred P (V.map (subst_term σ) args)
  | FNeg ψ => FNeg (subst_formula σ ψ)
  | FImp ψ ξ => FImp (subst_formula σ ψ) (subst_formula σ ξ)
  | FAll ψ => FAll (subst_formula (up_term_term σ) ψ)
  end.
```

Konačno, potrebno je definirati sintaksu za indukciju. U Coqu su definicije induktivnih propozicija proizvoljne do na ograničenje pozitivnosti, no radi jednostavnosti u FOL_{ID} su moguće samo induktivne definicije s atomarnim formulama, a pišemo ih u stilu prirodne dedukcije:

$$\frac{Q_1\mathbf{u}_1 \dots Q_n\mathbf{u}_n \qquad P_1\mathbf{v}_1 \dots P_m\mathbf{v}_m}{P\mathbf{t}}$$

Ovdje su $Q_1, \dots, Q_n$ obični predikatni simboli, $P_1, \dots, P_m$ i P su induktivni predikatni simboli, a podebljani znakovi predstavljaju vektore terma odgovarajućih duljina.

Definicija 7 (Produkcija)**.** Uređene četvorke

1. listi parova običnih predikatnih simbola i vektora terma odgovarajućih duljina,
2. listi parova induktivnih predikatnih simbola i vektora terma odgovarajućih duljina,
3. induktivnog predikatnog simbola P mjesnosti n i
4. vektora terma duljine n

nazivamo produkcijama.

```
Record production := mkProd {
    preds : list ({ P : PredS Σ & vec (term Σ) (pred_ar P) });
    indpreds : list ({ P : IndPredS Σ & vec (term Σ) (indpred_ar P) });
    indcons : IndPredS Σ;
    indargs : vec (term Σ) (indpred_ar indcons);
  }.
```

Skup induktivnih definicija je skup produkcija.

```
Definition IndDefSet := production -> Prop.
```

3.2. Semantika

Definicija 8 (Struktura)**.** Struktura prvog reda M je uređena četvorka skupa kojeg nazivamo *nosačem* te interpretacija funkcijskih, običnih predikatnih i induktivnih predikatnih simbola. Funkcijski se simboli interpretiraju kao n-mjesne funkcije, a predikatni simboli kao n-mjesne relacije na nosaču. Koristit ćemo ime strukture kao sinonim za njen nosač, a interpretacije označavamo sa f^M odnosno P^M.

```
Structure structure := {
    domain :> Set;
    interpF (f : FuncS Σ) : vec domain (fun_ar f) -> domain;
    interpP (P : PredS Σ) : vec domain (pred_ar P) -> Prop;
    interpIP (P : IndPredS Σ) : vec domain (indpred_ar P) -> Prop;
  }.
```

Definicija 9. Neka je M proizvoljna struktura. *Okolina* ρ za M je proizvoljna funkcija iz skupa prirodnih brojeva u nosač strukture.

```
Definition env := var -> M.
```

Okolina se može interpretirati kao niz $d_0, d_1, d_2, \ldots$. Tada je *pomaknuta okolina*, u oznaci $d \cdot \rho$, niz $d, d_0, d_1, d_2, \ldots$ za neki $d \in M$. Proširenje domene okoline ρ na skup terma zovemo *evaluacijom*.

```
Fixpoint eval (ρ : env) (t : term Σ) : M :=
  match t with
  | var_term x => ρ x
  | TFunc f args => interpF f (V.map (eval ρ) args)
  end.
```

Definicija 10 (Relacija ispunjivosti)**.** Neka je M proizvoljna struktura te ρ okolina za M. Ispunjivost formule φ u okolini ρ pišemo $\rho \vDash \varphi$, a definiramo rekurzivno na način:

1. ako je P običan ili induktivan predikatni simbol mjesnosti n te su $u_1, \ldots, u_n$ termi, onda vrijedi $\rho \vDash P(u_1, \ldots, u_n)$ ako i samo ako vrijedi $P^M(\rho(u_1), \ldots, \rho(u_n))$,
2. vrijedi $\rho \vDash \neg\varphi$ ako i samo ako ne vrijedi $\rho \vDash \varphi$,
3. vrijedi $\rho \vDash \varphi \to \psi$ ako i samo ako $\rho \vDash \varphi$ povlači $\rho \vDash \psi$ i
4. vrijedi $\rho \vDash \forall\varphi$ ako i samo ako za sve $d \in M$ vrijedi $d \cdot \rho \vDash \varphi$

```
Fixpoint Sat (ρ : env M) (F : formula Σ) : Prop :=
  match F with
    | FPred P args => interpP P (V.map (eval ρ) args)
    | FIndPred P args => interpIP P (V.map (eval ρ) args)
    | FNeg G => ~ Sat ρ G
    | FImp F G => Sat ρ F -> Sat ρ G
    | FAll G => forall d, Sat (d .: ρ) G
  end.
```

Lema 1. *Sintaktička i semantička supstitucija komutiraju pod relacijom ispunjivosti.*

```
Lemma strong_form_subst_sanity2 :
  forall (φ : formula Σ) (σ : var -> term Σ)
      (M : structure Σ) (ρ : env M),
      ρ ⊨ (subst_formula σ φ) <-> (σ >> eval ρ) ⊨ φ.
```

Misao: *Zvuči kul. Je li jasno što je to semantička supstitucija i zašto kažem da komutiraju?*

3.3. Standardni modeli

Operator φ_Φ. Aproksimanti. Standardni model.

3.4. Sistem sekvenata s induktivnim definicijama

LKID. Dopustiva pravila. Primjeri dokaza.

3.5. Adekvatnost

Lokalne adekvatnosti za pravila izvoda. Glavni teorem.

4. Ciklički dokazi

Koinduktivni tip podatka i koinduktivna propozicija. Jedan primjer su `Streamovi` i predikat `Infinite`. Jednostavniji primjer bi možda bio koinduktivni `nat` i koinduktivni `le`.

Kako bi izgledali ciklički dokazi u LKID? Ono što je tamo "repeat funkcija" je u Coqu `cofix`.

5. Zaključak

Literatura

[1] Y. Bertot i P. Castéran, *Interactive theorem proving and program development: Coq'Art: the Calculus of Inductive Constructions*. Springer Science & Business Media, 2013.

[2] B. C. Pierce, A. A. de Amorim, C. Casinghino, M. Gaboardi, M. Greenberg, C. Hriţcu, V. Sjöberg, i B. Yorgey, *Logical Foundations*, ser. Software Foundations, B. C. Pierce, Ur. Electronic textbook, 2023., sv. 1, version 6.5, http://softwarefoundations.cis.upenn.edu.

[3] B. C. Pierce, A. A. de Amorim, C. Casinghino, M. Gaboardi, M. Greenberg, C. Hriţcu, V. Sjöberg, A. Tolmach, i B. Yorgey, *Programming Language Foundations*, ser. Software Foundations, B. C. Pierce, Ur. Electronic textbook, 2024., sv. 2, version 6.5, http://softwarefoundations.cis.upenn.edu.

[4] A. W. Appel, *Verified Functional Algorithms*, ser. Software Foundations, B. C. Pierce, Ur. Electronic textbook, 2023., sv. 3, version 1.5.4, http://softwarefoundations.cis.upenn.edu.

[5] A. Chlipala, *Certified programming with dependent types: a pragmatic introduction to the Coq proof assistant*. MIT Press, 2022.

[6] The Coq Development Team, "The Coq Reference Manual, Release 8.18.0", https://coq.inria.fr/doc/v8.18/refman/, 2023.

[7] F. Pfenning i C. Paulin-Mohring, " Inductively Defined Types in the Calculus of Constructions", u *Proceedings of the 5th International Conference on Mathematical Foundations of Programming Semantics*. Berlin, Heidelberg: Springer-Verlag, 1989., str. 209–228.

[8] E. Giménez, “Codifying guarded definitions with recursive schemes”, u *Types for Proofs and Programs*, P. Dybjer, B. Nordström, i J. Smith, Ur. Berlin, Heidelberg: Springer Berlin Heidelberg, 1995., str. 39–59.

[9] M. Sozeau, S. Boulier, Y. Forster, N. Tabareau, i T. Winterhalter, “Coq Coq correct! verification of type checking and erasure for Coq, in Coq”, *Proceedings of the ACM on Programming Languages*, sv. 4, br. POPL, str. 1–28, 2019.

[10] G. Hutton, *Programming in Haskell*, 2. izd. Cambridge University Press, 2016.

[11] J. Brotherston, “Cyclic proofs for first-order logic with inductive definitions”, u *Automated Reasoning with Analytic Tableaux and Related Methods*, B. Beckert, Ur. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005., str. 78–92.

[12] ——, “Sequent calculus proof systems for inductive definitions”, doktorska disertacija, School of Informatics, University of Edinburgh, 2006.

[13] N. G. de Bruijn, “Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem”, *Indagationes Mathematicae (Proceedings)*, sv. 75, br. 5, str. 381–392, 1972.

[14] B. C. Pierce, *Types and Programming Languages*. MIT Press, 2002.

[15] K. Stark, “Mechanising Syntax with Binders in Coq”, doktorska disertacija, Saarland University, 2020.

[16] K. Stark, S. Schäfer, i J. Kaiser, “Autosubst 2: Reasoning with Multi-Sorted de Bruijn Terms and Vector Substitutions”, *8th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2019, Cascais, Portugal, January 14-15, 2019*, 2019.

Sažetak

Primjene Coq alata za dokazivanje u matematici i računarstvu

Miho Hren

Unesite sažetak na hrvatskom.

Ključne riječi: prva ključna riječ; druga ključna riječ; treća ključna riječ

Abstract

Applications of the Coq Proof Assistant in mathematics and computer science

Miho Hren

Enter the abstract in English.

Keywords: the first keyword; the second keyword; the third keyword