Ovo je zahvala.

Sadržaj

1. Uvod

2. Coq

U ovom poglavlju dajemo pregled visoke razine 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, a može se koristiti i kao funkcijski programski jezik 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 ljuskama `bash` i `python`. Interaktivna ljuska (također poznata pod imenom *toplevel*) služi unosu definicija te iskazivanju i dokazivanju tvrdnji. Skupni način rada pokreće se naredbom `coqc`, a korisniku omogućuje semantičku provjeru i prevođenje izvornih datoteka u strojno čitljive formate. Kod formaliziranja i dokazivanja, korisnik će najčešće koristiti interaktivni način rada, po mogućnosti kroz neku od dostupnih razvojnih okolina. [2]

[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*.

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 iskazivanje tvrdnji, kao što je `Lemma`, Coq ulazi u način dokazivanja (*proof mode*).

Gallina je Coqov strogo statički tipiziran specifikacijski jezik. Dokazi svih tvrdnji predstavljeni su interno kao programi u Gallini. Kako se glavnina programiranja u Coqu svodi upravo na programiranje u Gallini, posvećujemo joj idući odjeljak.

Ltac je Coqov netipizirani 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 je programskog sustava Coq algoritam za provjeru tipova (*type checking*) – svaka tvrdnja koja se dokazuje iskazana je tipovima. Ostatak sustava u načelu služi za knjigovodstvo i poboljšanje korisničkog iskustva.[3] 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?

Prve inačice Coqa implementirale su samo račun konstrukcija [7] – proširenje λ-računa polimorfnim i zavisnim tipovima te tipskim konstruktorima. Kasnije je dodana podrška za induktivno i koinduktivno definirane tipove [8, 9], a danas se može reći da Coq implementira polimorfni kumulativni račun induktivnih konstrukcija [10]. Coq se, osim kao dokazivač teorema, može koristiti i za programiranje sa zavisnim tipovima. U

[3] I jezgra i ostatak sustava implementirani su u OCamlu.

toj sferi konkuriraju jezici Agda[4], Idris[5] i Lean[6]. 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 prevesti u jezik niže razine apstrakcije.[7] Mehanizam ekstrakcije nije dokazano točan, no poželjno je da izvorne funkcije budu ekvivalentne ekstrahiranima pa se radi na verifikaciji ekstrakcije [10].

2.2. Programiranje u Coqu

Gallina je funkcijski programski jezik, što znači da su funkcije prvoklasni objekti — one mogu biti argumenti i povratne vrijednosti drugih funkcija. Dodatno, iteracija se ostvaruje rekurzijom te ne postoje tradicionalne varijable, već se koriste nepromjenjiva (*immutable*) imena. Za uvod u funkcijsko programiranje, čitatelja upućujemo na knjigu *Programming in Haskell* [11]. Primjeri koje ćemo vidjeti u ostatku ovog odjeljka oslanjanju se 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 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".

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

- `forall` za konstrukciju funkcijskih tipova i zavisnih produkata,
- `match` za rad sa stanovnicima induktivnih tipova te
- `fun`, `fix` i `cofix` za definiciju funkcija.

Naredbom `Inductive` definira se *induktivni* tip te se automatski za njega generiraju

[4] `https://wiki.portal.chalmers.se/agda/`
[5] `https://www.idris-lang.org/`
[6] `https://lean-lang.org/`
[7] Trenutno su podržani Haskell, OCaml i Scheme.
[8] `https://coq.inria.fr/library/`
[9] Tipovi su kolekcije objekata na kojima je moguće provoditi srodne operacije.

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*).

Jedna od osnovnih naredbi za imenovanje 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 izrazi koji počinju s `match t`, gdje je `t` stanovnik tipa `T`, moraju imati po jednu granu za svaki konstruktor tipa `T`.[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 tvrdnja univerzalno kvantificirana po prirodnim brojevima.

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

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

```
Fixpoint plus (n m : nat) {struct n} : nat :=
(* Definition plus := fix plus (n m : nat) {struct n} := *)
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 s obzirom na prvi argument što je vidljivo iz oznake `{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) := Cons {
      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 naredbu `Definition` te izraz `cofix`.

```
CoFixpoint from (n : nat) : Stream nat := Cons _ n (from (n + 1)).
(* Definition from := cofix from (n : 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 sumirati epigramom:

> „Induktivni tipovi su domene rekurzivnih funkcija, koinduktivni tipovi su kodomene korekurzivnih funkcija”.

Time se želi reći da se termi induktivnih tipova destrukturiraju u rekurzivnim funkcijama, dok se termi koinduktivnih tipova konstruiraju u korekurzivnim funkcijama.

2.3. Hijerarhija tipova

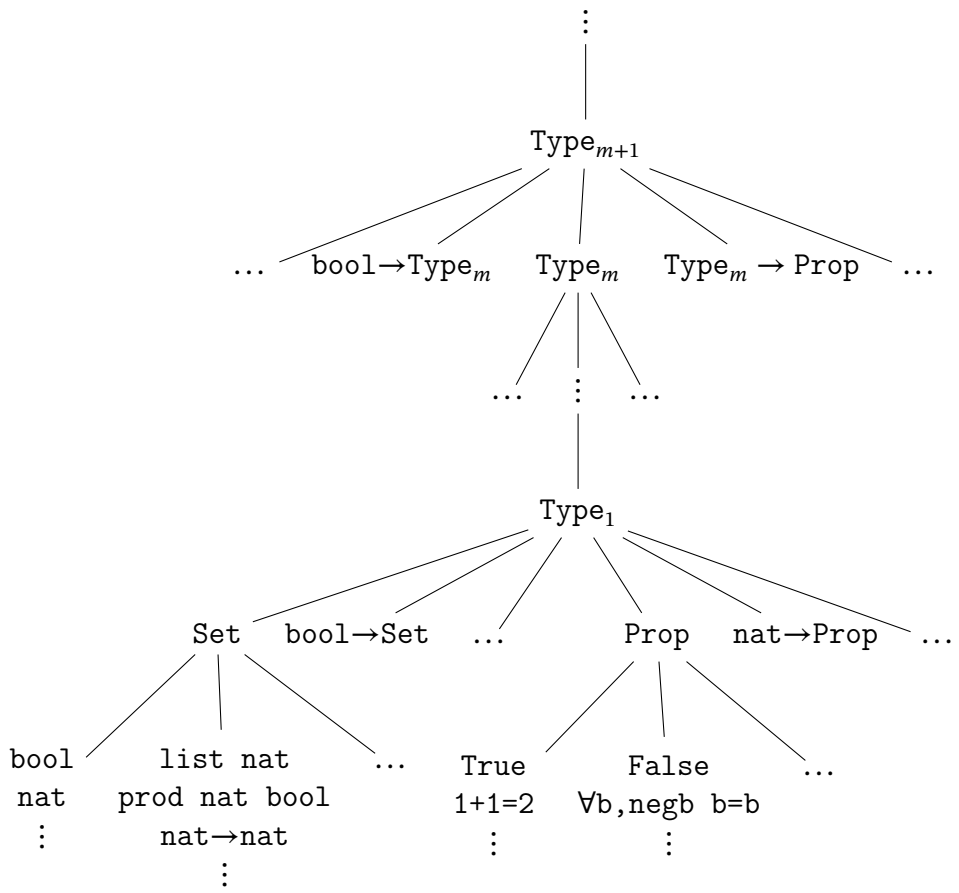
U usporedbi s tradicionalnim programskim jezicima, Coqov tipski sustav je ekspresivniji jer dopušta tipove koji mogu ovisiti o termima. Takvi tipovi se u Coqu konstruiraju izrazom oblika `forall`. Primjer jednog takvog tipa je „lista duljine n", gdje je n neki prirodan broj. Njegovi su stanovnici n-torke,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

Spomenuli smo da su i tipovi termi te im se može dodijeliti sorta. Postoji li najveća sorta, odnosno postoji li tip `Type` čiji su stanovnici svi tipovi? Prisjetimo se, svaki term ima svoj tip. Kada bi takav `Type` postojao, tada bi vrijedilo `Type : Type`, što može dovesti do paradoksa samoreferenciranja.[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 rastući niz 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 `Set` i `Prop`.

Naziv `Set` sinonim je za sortu $\texttt{Type}_0$, a njeni stanovnici su **mali tipovi**. **Misao:** *Ostavio sam naziv „mali tipovi" jer kasnije kažem da za `Set` nema kvantifikacije po velikim tipovima, pa da zadržim taj kontrast.* Primjerice, tipovi `nat` i `bool` su mali tipovi. Dodatno, tipovi funkcija koje primaju i vraćaju male tipove su također mali tipovi. Također su i produkti, sume, liste i stabla malih tipova ponovo 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** (izjave). Za razliku od programa, s propozicijama ne možemo efektivno računati, ali ih možemo dokazivati. Stanovnici propozicija su njihovi **dokazi**.

[11] U naivnoj logici to je Epimenidov paradoks („Ova je rečenica lažna."), u naivnoj teoriji skupova to je Russellov paradoks, u naivnoj teoriji tipova to je Girardov paradoks.



Slika 2.1. Kumulativna hijerarhija tipova

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. Posljedično, svaki term koji u sebi sadrži `Set` ili `Type` nužno nije u `Set`. Na praktičnoj strani, predikativnost ograničava korisnika da prilikom definicije programa smije kvantificirati

samo po malim tipovima, odnosno da 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 pojavljivala samo konačno mnogo puta, broj π bio bi racionalan. Međutim, nije jasno *koja* znamenka ima to svojstvo. Možda ih ima više. Štoviše, vjerujemo da su *sve* znamenke takve. Da bismo odgovorili na to pitanje, morali bismo prebrojiti sve znamenke 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 njen argument i pomoću njega konstruirati povratnu vrijednost. Dodatno, nemoguće je dokazati laž, a istina trivijalno vrijedi. S druge strane, ako induktivni tip nema konstruktore, onda je nenastanjen jer nije moguće definirati vrijednost tog tipa. Ako pak tip ima barem jedan konstruktor bez argumenata, 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 na prirodnim brojevima, princip matematičke indukcije glasi:

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

Tvrdnju dokazujemo analizom broja n. Ako je $n = 0$, tvrdnja slijedi iz *baze* indukcije.

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	zavisna suma

Tablica 2.1. Sličnosti dokazivanja i programiranja

Ako je pak $n = n' + 1$ 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
    | O => baza
    | S n' => korak n' (F n')
    end.
```

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

2.5. Ograničenja tipskog sustava

Kao što smo već vidjeli, Coqov tipski sustav je izražajniji od tipskih sustava uobičajenih 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. 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 (funkcija se poziva samo na pravom podtermu originalnog argumenta).

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 pojaviti samo kao izravni argument konstruktora koinduktivnog tipa čiji element definiramo (poziv funkcije mora stvoriti pravi nadterm originalnog poziva).[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.

Posljednje ograničenje koje spominjemo vezano je uz irelevantnost dokaza (*proof irrelevance*). Naime, mnogi teoremi mogu se dokazati na više načina, ali pojedini dokaz (dakle, postupak kojim smo od pretpostavki došli do konkluzije) *nije bitan*. Matematičarima su bitni samo iskaz teorema i činjenica da se teorem *može dokazati*. Sam postupak dokazivanja smatra se „implementacijskim detaljem". 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 omogućavanja 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, jer bi prilikom ekstrakcije bilo nužno zadržati i sve dokaze uz svu kompleksnost njihovih tipova.

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 [12]. Definicije, leme i dokazi u ovom poglavlju preuzete su iz Brotherstonove disertacije [13]. Za općeniti uvod u logiku čitatelja upućujemo na knjigu *Matematička logika* [14].

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

[1] TODO: REPO LINK

domene diskursa formaliziraju tipovima sorte `Set`[2], dok se skupovi kao predikati formaliziraju 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 svih 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 $f(t_1, \ldots, t_n)$

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

[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.

također term.

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

Općenitije prezentacije logike prvog reda za skup varijabli uzimaju proizvoljan skup $\mathcal{V}$, no za formalizaciju je pogodniji skup prirodnih brojeva $\mathbb{N}$. Kasnije ćemo objasniti zašto.

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 svih 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.

[5] Ovdje `vec`.

```
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.
```

Ostale veznike definiramo kao sintaksne pokrate.

```
Definition FAnd (φ ψ : formula) : formula := FNeg (FImp φ (FNeg ψ)).
Definition FOr (φ ψ : formula) : formula := FImp (FNeg φ) ψ.
Definition FExist (φ : formula) : formula := FNeg (FAll (FNeg φ)).
```

Umjesto eksplicitne kvantifikacije po nekoj varijabli v, 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 [15]. O samoj implementaciji de Bruijnovog indeksiranja više se može pročitati u knjizi *Types and Programming Languages* [16]. Za potrebe ovog rada koristili smo program *Autosubst2*[7] [17, 18] 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 svih terma.

[6] Ili općenitije, vezivanju varijabli.

[7] `https://github.com/uds-psl/autosubst2`

Supstituciju σ možemo promatrati kao niz terma $t_0, t_1, t_2, ...$ Tada je *pomaknuta supstitucija*, u oznaci $t \cdot \sigma$, supstitucija koja odgovara nizu $t, t_0, t_1, t_2, ...$

Domena supstitucije može se rekurzivno proširiti na skup svih terma i skup svih 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.
```

```
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.
```

Ovdje funkcija `up_term_term` brine da supstitucija σ mijenja samo one varijable koje nisu vezane. Pišemo $\varphi[\sigma]$ za primjenu supstitucije σ na formulu φ. Posebno, pišemo $\varphi[t/x]$ za supstituciju varijable x termom t u formuli φ te pišemo $\varphi^{\uparrow}$ za inkrementaciju svake varijable u formuli φ. Iste notacije koristimo i za supstitucije na termima, listama terma i listama formula.

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 n-torke terma, gdje je n mjesnost odgovarajućeg predikata.

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

1. listi parova običnih predikatnih simbola i n-torki terma odgovarajućih duljina,

2. listi parova induktivnih predikatnih simbola i n-torki terma odgovarajućih duljina,
3. induktivnog predikatnog simbola P mjesnosti m i
4. m-torke terma

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 Φ je skup produkcija. U ostatku rada promatramu neki proizvoljan, ali fiksiran skup induktivnih definicija Φ.

```
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 svih 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.
```

Pišemo t^{ρ} za evaluaciju terma t u okolini ρ. Istu notaciju koristimo i za evaluaciju n-torki terma.

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

Želimo ograničiti semantička razmatranja na samo one strukture koje „imaju smisla” za induktivne predikate. Prisjetimo se, predikatni simbol P mjesnosti n interpretira se na

strukturi M podskupom skupa M^n. Indukciju smatramo dokazivanjem u razinama pa ima smisla promatrati *razine interpretacije* induktivnog predikata, gdje je nulta razina prazan skup, a svaku iduću razinu konstruiramo pomoću produkcija induktivnog skupa definicija i prethodnih razina. Tako je prva razina onaj podskup kojeg možemo dobiti samo jednom „primjenom produkcija", druga je razina onaj podskup kojeg možemo dobiti pomoću dvije primjene produkcija, i tako dalje. Na taj se način, korak po korak, gradi *smislena* interpretacija induktivnih predikata. Napominjemo da se zbog mogućih međuovisnosti induktivnih predikata razine interpretacije definiraju simultano. Ovaj odjeljak posvećujemo formalizaciji ovih pojmova.

Definicija 11 (Operator skupa definicija)**.** Neka je M proizvoljna struktura te neka je pr proizvoljna produkcija induktivnog skupa definicija Φ u čijoj se konkluziji javlja induktivni predikatni simbol P, primjerice:

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

Neka je f funkcija koja svakom induktivnom predikatnom simbolu R pridružuje podskup skupa $M^{|R|}$. Tada je $\varphi_{pr}(f)$ skup svih $|P|$-torki $\mathbf{d}$ elemenata nosača M za koje postoji okolina ρ za M takva da:

- za sve $i \in \{1, \dots, n\}$ vrijedi $\mathbf{u}_i^\rho \in Q_i^M$,
- za sve $j \in \{1, \dots, m\}$ vrijedi $\mathbf{v}_\mathbf{i}^\rho \in f(P_i)$ i
- $\mathbf{d} = \mathbf{t}^\rho$.

```
1  Definition φ_pr
2    (pr : production)
3    (args : forall P : IndPredS Σ, vec D (indpred_ar P) -> Prop)
4    (ds : vec D (indpred_ar (indcons pr)))
5    : Prop :=
6        exists (ρ : env M),
7        (forall Q us, List.In (Q; us) (preds pr) ->
8                  interpP Q (V.map (eval ρ) us))
9        /\
10         ( forall P ts, List.In (P; ts) (indpreds pr) ->
11                   args P (V.map (eval ρ) ts))
12       /\
13         ds = V.map (eval ρ) (indargs pr).
```

Operator φ_{pr} je formalizacija ideje primjene produkcije. Nadalje, potrebno je definirati operator koji će uzeti u obzir *sve* produkcije koje se odnose na P. Definiramo $\varphi_P(f)$ kao

uniju svih $\varphi_{pr'}(f)$ gdje je pr' produkcija u kojoj se P javlja u konkluziciji.

```
Definition φ_P
  (P : IndPredS Σ)
  (args : forall P : IndPredS Σ, vec D (indpred_ar P) -> Prop)
  : vec D (indpred_ar P) -> Prop.
  refine (fun ds => _).
  refine (@ex production (fun pr => _)).
  refine (@ex (P = indcons pr /\ Φ pr) (fun H => _)).
  destruct H as [Heq HΦ].
  rewrite Heq in ds.
  exact (φ_pr pr args ds).
Defined.
```

Konačno, definiramo operator skupa definicija φ_Φ kao preslikavanje koje induktivnom predikatnom simbolu P pridružuje skup $\varphi_P(f)$.

```
Definition φ_Φ
  (args : forall P : IndPredS Σ, vec D (indpred_ar P) -> Prop)
    : forall P : IndPredS Σ, vec D (indpred_ar P) -> Prop :=
    fun P => φ_P P args.
```

Operator φ_Φ omogućuje simultanu primjenu produkcija.

Napomena. Kako je funkcija f bila uvedena na samom početku prethodne definicije, u stvari definicija operatora φ_Φ glasi $\varphi_\Phi(f)(P) \coloneqq \varphi_P(f)$.

Misao: *Ova duga definicija vapi za primjerom. Meni je jasno, ali slutim da bi dragom čitatelju bilo incomprehensible.*

Propozicija 1. *Operator φ_Φ je monoton.*

```
Proposition φ_Φ_monotone :
  forall (f g : forall P, vec D (indpred_ar P) -> Prop),
    (forall P v, f P v -> g P v) ->
    (forall P v, φ_Φ f P v -> φ_Φ g P v).
```

Definicija 12 (Aproksimanti)**.** Neka je M proizvoljna struktura. Definiramo aproksimaciju skupa induktivnih definicija Φ razine α, u oznaci φ_Φ^α, rekurzivno na način:

1. $\varphi_\Phi^0(P) \coloneqq \varnothing$ i
2. $\varphi_\Phi^{\alpha+1} \coloneqq \varphi_\Phi(\varphi_\Phi^\alpha)$.

```
Fixpoint φ_Φ_n P (α : nat) (v : vec M (indpred_ar P)) : Prop :=
  match α with
  | 0 => False
  | S α => @φ_Φ Σ M Φ (fun P => φ_Φ_n P α) P v
  end.
```

Tada je aproksimant induktivnog predikatnost simbola P razine α upravo $\varphi^{\alpha}_{\Phi}(P)$.

```
Definition approximant_of (P : IndPredS Σ)
  : nat -> vec M (indpred_ar P) -> Prop :=
  φ_Φ_n P.
```

Misao: *Redoslijed argumenata u approximant_of je zbunjujuć s obzirom na čitanje simbola: fi fi na alfa od pe.* **Misao:** *O redoslijedu smo na nekom sastanku raspravljali i zaključili da bi ipak trebao biti ovakav kakav jest.*

Napomena. Brotherston je definirao aproksimaciju razine α kao uniju aproksimacija svih nižih razina. Takva je definicija ekvivalentna našoj.

Misao: *Malo je neprezicno reći "unija", ustvari se misli na uniju za svaki pojedini P.*

Lema 2. *Za sve prirodne brojeve α i induktivne predikatne simbole P vrijedi*

$$\varphi^{\alpha}_{\Phi}(P) = \bigcup_{\beta<\alpha} \varphi^{\beta}_{\Phi}(P).$$

```
Lemma approximant_characterization : forall a P v,
      φ_Φ_n P a v <-> exists β,
                        β < a /\ @φ_Φ Σ M Φ (fun P => φ_Φ_n P β) P v.
```

Definicija 13. Aproksimacija razine ω, u oznaci φ^{ω}_{Φ}, je unija aproksimacija razina manjih od ω.

```
Definition φ_Φ_ω P v := exists α, φ_Φ_n P α v.
```

Misao: *Opet, "unija" nije precizno.*

Lema 3. *Aproksimacija razine ω najmanji je skup sa svojstvom $\varphi_{\Phi}(\varphi^{\omega}_{\Phi}) \subseteq \varphi^{\omega}_{\Phi}$.*

```
Lemma ω_prefixed : forall P υ, @φ_Φ Σ M Φ φ_Φ_ω P υ -> φ_Φ_ω P υ.
Lemma ω_least : forall args,
                (forall P υ, @φ_Φ Σ M Φ args P υ -> args P υ) ->
                forall P υ, φ_Φ_ω P υ -> args P υ.
```

Definicija 14 (Standardni model)**.** Kažemo da je struktura M standardni model za Φ ako interpretira svaki induktivni predikatni simbol njegovim aproksimantom razine ω.

```
Definition standard_model
  (Σ : signature) (Φ: @IndDefSet Σ)
  : structure Σ -> Prop :=
  fun M =>
    forall (P : IndPredS Σ) ts, interpIP P ts <-> @φ_Φ_ω Σ M Φ P ts.
```

Standardni modeli su upravo one strukture koje imaju smisla za skup induktivnih definicija.

3.4. Sistem sekvenata s induktivnim definicijama

Ovaj odlomak je za diskusiju na sastanku. Kako to da je Brotherston definirao signaturu, ali onda nije uključio aksiome u LKID? Primjerice, u Σ_{PA} imamo simbol jednakosti, ali ne možemo dokazati refleksivnost. Valjda su aksiomi uključeni u Γ, u smislu LKID je općenit dokazni sustav, pa bi neka "aksiomatizirana teorija" u stvari bili dokazi *akiomi* $\vdash \varphi$. Možda.

Cilj je ovog odjeljka definirati dokazni sustav *LKID* za logiku FOL_{ID}. Ovaj dokazni sustav temelji se na Gentzenovom računu sekvenata, a proširen je lijevim i desnim pravilima za indukciju. Prije no što definiramo *LKID*, potrebno je definirati pojmove sekvenata i međusobnih zavisnosti induktivnih predikata.

Definicija 15 (Sekvent)**.** Neka su Γ i Δ proizvoljne liste formula. Tada je sekvent uređeni par (Γ, Δ), a označavamo ga s $\Gamma \vdash \Delta$.

```
Inductive sequent : Set :=
| mkSeq (left : list (formula Σ)) (right : list (formula Σ)).
```

Sekventom $\Gamma \vdash \Delta$ konceptualno tvrdimo da istinitost *svake* tvrdnje u Γ povlači istinitost *neke* tvrdnje u Δ. Kažemo da je sekvent $\Gamma \vdash \Delta$ *dokaziv*, ili da iz Γ postoji izvod za Δ, ako u sustavu *LKID* postoji dokaz za $\Gamma \vdash \Delta$. Kasnije ćemo precizirati što je točno *dokaz*.

Definicija 16 (Relacija *Prem*)**.** Neka su P_i i P_j proizvoljni induktivni predikatni simboli. Kažemo da je simbol P_i u relaciji *Prem* sa simbolom P_j ako postoji produkcija u skupu induktivnih definicija Φ takva da se simbol P_i javlja u njenom konzekvensu te se simbol P_j javlja u njenim premisama.

```
Definition Prem (Pi Pj : IndPredS Σ) :=
  exists pr, Φ pr /\
             indcons pr = Pi /\
             exists ts, List.In (Pj; ts) (indpreds pr).
```

Definicija 17 (Međusobna zavisnost)**.** Definiramo relaciju $Prem^*$ kao refleksivno i tranzitivno zatvorenje relacije *Prem*.

```
Definition Prem_star := clos_refl_trans (IndPredS Σ) Prem.
```

Tada za induktivne predikatne simbole P i Q kažemo da su međusobno zavisni ako vrijedi $Prem^*(P,Q)$ i $Prem^*(Q,P)$.

```
Definition mutually_dependent (P Q : IndPredS Σ) :=
  Prem_star P Q /\ Prem_star Q P.
```

Lema 4. *Međusobna zavisnost je relacija ekvivalencije na skupu induktivnih predikatnih simbola.* ***Misao:*** *Ova lema zvuči kao da je bitna, ali čini mi se da nije.*

```
Lemma mutually_dependent_refl : forall P, mutually_dependent P P.
Lemma mutually_dependent_symm : forall P Q,
      mutually_dependent P Q ->
      mutually_dependent Q P.
Lemma mutually_dependent_trans : forall P Q R,
      mutually_dependent P Q ->
      mutually_dependent Q R ->
      mutually_dependent P R.
```

Sada ćemo definirati sustav *LKID* u četiri koraka. U prva tri koraka ćemo definirati strukturalna, propozicijska i kvantifikatorska pravila izvoda. U zadnjem koraku ćemo definirati pravila izvoda za indukciju. Sve varijable koje se javljaju u pravilima izvoda su implicitno univerzalno kvantificirana.

Misao: *Rado bih koristio nešto što nije `figure` za pravila izvoda, ali ne znam što.*

Strukturalna pravila su prikazana u slici 3.1., a služe baratanju strukture sekvenata, dok ostala pravila služe logičkom zaključivanju. Pravilo *Ax* kaže da je sekvent dokaziv

ako se ista formula nalazi s lijeve i s desne strane znaka $\vdash$. Pravilo Wk, ili pravilo *slabljenja*, kaže da je uvijek moguće dokazati tvrdnje koje su slabije od već dokazanih. Drugim riječima, dodavanje formula u pretpostavke ili zaključak ne mijenja dokazivost. Pravilo Cut je svojevrsni *modus ponens*. Konačno, pravilo $Subst$ kaže da se u već dokazanom sekventu varijable mogu proizvoljno mijenjati.

$$\frac{\Gamma \cap \Delta \neq \emptyset}{\Gamma \vdash \Delta}\,(Ax)$$

$$\frac{\Gamma' \vdash \Delta' \qquad \Gamma' \subseteq \Gamma \qquad \Delta' \subseteq \Delta}{\Gamma \subseteq \Delta}\,(Wk)$$

$$\frac{\Gamma \vdash \varphi, \Delta \qquad \varphi, \Gamma \vdash \Delta}{\Gamma \vdash \Delta}\,(Cut)$$

$$\frac{\Gamma \vdash \Delta}{\Gamma[\sigma] \vdash \Delta[\sigma]}\,(Subst)$$

Slika 3.1. Strukturalna pravila sustava $LKID$.

Propozicijska pravila prikazana su na slici 3.2., a odnose se na propozicijski dio logike FOL_{ID}, odnosno na negaciju i implikaciju. **Misao:** *Treba li tu neka motivacija? Ja bih rekao da ne.*

$$\frac{\Gamma \vdash \varphi, \Delta}{\neg\varphi, \Gamma \vdash \Delta}\,(NegL)$$

$$\frac{\varphi, \Gamma \vdash \Delta}{\Gamma \vdash \neg\varphi, \Delta}\,(NegR)$$

$$\frac{\Gamma \vdash \varphi, \Delta \qquad \psi, \Gamma \vdash \Delta}{\varphi \to \psi, \Gamma \vdash \Delta}\,(ImpL)$$

$$\frac{\varphi, \Gamma \vdash \psi, \Delta}{\Gamma \vdash \varphi \to \psi, \Delta}\,(ImpR)$$

Slika 3.2. Propozicijska pravila sustava $LKID$.

Kvantifikacijska pravila prikazana su na slici 3.3., a odnose se na univerzalnu kvantifikaciju. Supstitucija σ_{id} je supstitucija identiteta. Lijevo pravilo za kvantifikaciju kaže da dokaz koji počinje s instanciranom formulom φ u kojoj se javlja term t na mjestu varijable 0 isto tako smije započeti s univerzalno kvantificiranom formulom $\forall\varphi$. Za desno pravilo kvantifikacije primijetimo da se varijabla 0 ne javlja u listama $\Gamma^{\uparrow}$ i $\Delta^{\uparrow}$ zbog pomicanja. Kako naše formule zbog de Bruijnovog indeksiranja implicitno kvantificiraju po

varijabli 0, a varijabla 0 se možda javlja u formuli φ, ustvari je dokaz sekventa $\Gamma \vdash \forall\varphi, \Delta$ posve analogan dokazu sekventa $\Gamma^{\uparrow} \vdash \varphi, \Delta$. **Misao:** *Ovo vjerojatno treba preformulirati.*

$$\frac{\varphi[t \cdot \sigma_{id}], \Gamma \vdash \Delta}{\forall\varphi, \Gamma \vdash \Delta}\,(AllL)$$

$$\frac{\Gamma^{\uparrow} \vdash \varphi, \Delta^{\uparrow}}{\Gamma \vdash \forall\varphi, \Delta}\,(AllR)$$

Slika 3.3. Kvantifikacijska pravila sustava *LKID*.

```
Inductive LKID : sequent -> Prop :=
  (* Structural rules. *)
  | Ax : forall Γ Δ φ, In φ Γ -> In φ Δ -> LKID (Γ ⊢ Δ)
  | Wk : forall Γ' Δ' Γ Δ,
      Γ' ⊆ Γ ->
      Δ' ⊆ Δ ->
      LKID (Γ' ⊢ Δ') ->
      LKID (Γ ⊢ Δ)
  | Cut : forall Γ Δ φ,
      LKID (Γ ⊢ φ :: Δ) ->
      LKID (φ :: Γ ⊢ Δ) ->
      LKID (Γ ⊢ Δ)
  | Subst : forall Γ Δ,
      LKID (Γ ⊢ Δ) ->
      forall σ, LKID (map (subst_formula σ) Γ ⊢ map (subst_formula σ) Δ)
  (* Propositional rules. *)
  | NegL : forall Γ Δ φ, LKID (Γ ⊢ φ :: Δ) -> LKID (FNeg φ :: Γ ⊢ Δ)
  | NegR : forall Γ Δ φ, LKID (φ :: Γ ⊢ Δ) -> LKID (Γ ⊢ FNeg φ :: Δ)
  | ImpL : forall Γ Δ φ ψ,
      LKID (Γ ⊢ φ :: Δ) -> LKID (ψ :: Γ ⊢ Δ) ->
      LKID (FImp φ ψ :: Γ ⊢ Δ)
  | ImpR : forall Γ Δ φ ψ,
      LKID (φ :: Γ ⊢ ψ :: Δ) -> LKID (Γ ⊢ (FImp φ ψ) :: Δ)
  (* Quantifier rules. *)
  | AllL : forall Γ Δ φ t,
      LKID (subst_formula (t .: ids) φ :: Γ ⊢ Δ) ->
      LKID (FAll φ :: Γ ⊢ Δ)
  | AllR : forall Γ Δ φ,
      LKID (shift_formulas Γ ⊢ φ :: shift_formulas Δ) ->
      LKID (Γ ⊢ (FAll φ) :: Δ)
  | IndL : forall Γ Δ (Pj : IndPredS Σ) (u : vec (term Σ) (indpred_ar Pj))
             (z_i : forall P, vec var (indpred_ar P)) (* dodati pretpostavku foral
             (G_i : IndPredS Σ -> formula Σ)
             (HG2 : forall Pi, ~mutually_dependent Pi Pj -> G_i Pi = FIndPred Pi (
      let maxΓ := max_fold (map some_var_not_in_formula Γ) in
      let maxΔ := max_fold (map some_var_not_in_formula Δ) in
      let maxP := some_var_not_in_formula (FIndPred Pj u) in
      let shift_factor := max maxP (max maxΓ maxΔ) in
      let Fj := subst_formula (finite_subst (z_i Pj) u) (G_i Pj) in
      let minor_premises :=
        (forall pr (Hdep : mutually_dependent (indcons pr) Pj),
            let Qs := shift_formulas_by shift_factor (FPreds_from_preds (preds pr))
            let Gs := map (fun '(P; args) =>
                             let shifted_args := V.map (shift_term_by shift_factor)
                             let σ := finite_subst (z_i P) (shifted_args) in
                             let G := G_i P in
                             subst_formula σ G)
                          (indpreds pr) in
            let Pi := indcons pr in
            let ty := V.map (shift_term_by shift_factor) (indargs pr) in
            let Fi := subst_formula (finite_subst (z_i Pi) ty) (G_i Pi) in
            LKID (Qs ++ Gs ++ Γ ⊢ Fi :: Δ))
```

Za neko pravilo izvoda kažemo da je *dopustivo* ako nije u definiciji sustava $LKID$, ali je iz originalnih pravila izvedivo.

Primjer 2. Obično se kod definicije sistema sekvenata za varijacije logike prvog reda dodaju i pravila za egzistencijalnu kvantifikaciju. Lijevo i desno pravilo za egzistencijalnu kvantifikaciju je dopustivo u sustavu $LKID$.

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] T. Coquand i G. Huet, "The calculus of constructions", INRIA, teh. izv. RR-0530, svibanj 1986. [Mrežno]. Adresa: https://inria.hal.science/inria-00076024

[8] 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.

[9] 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.

[10] 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.

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

[12] 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.

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

[14] M. Vuković, *Matematička logika*. Element, 2009.

[15] 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.

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

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

[18] 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