Ovo je zahvala.

Sadržaj

1. Uvod

Coq je interaktivni dokazivač teorema u kojem korisnik iskazuje matematičke tvrdnje tipovima te ih dokazuje programiranjem. Teorija tipova koju Coq implementira temelji se na računu induktivnih konstrukcija koji pak je proširenje λ-računa tipovima. Iskazivanje tvrdnji moguće je zahvaljujući sorti `Prop` te zavisnim tipovima. Zahvaljujući jeziku taktika moguće je i automatizirano dokazivanje nekih tvrdnji, a osim za dokazivanje, Coq se može koristiti i za pisanje dokazano točnih programa.

Središnji pojam ovog rada je logika prvog reda s induktivnim definicijama FOL_{ID}, koju je prvi uveo James Brotherston, a u kojoj se neki predikati definiraju na induktivni način produkcijama. Zbog induktivnih predikata nužno je sa semantičkog stajališta promatrati posebne vrste struktura u kojima interpretacije induktivnih predikata imaju smisla s obzirom na njihove definicije. Sa sintaksnog stajališta promatra se Gentzenov sistem sekvenata proširen pravilima za induktivne predikate.

Ovaj rad sastoji se od četiri dijela. U prvom dijelu prikazujemo osnovne Coqa kao programskog jezika te osnove njegove teorijske pozadine, prvenstveno hijerarhije tipova i Curry–Howardove korespondencije. U drugom dijelu čitatelj se upoznaje s logikom FOL_{ID} kroz pojmove formula, struktura te standardnih modela. U trećem dijelu definiramo dokazni sustav $LKID$ za logiku FOL_{ID} te dokazujemo neka njegova svojstva. Sve definicije i tvrdnje u drugom i trećem dijelu popraćene su svojim formalizacijama u Coqu. Konačno, u četvrtom dijelu ilustriramo pojam cikličkog dokaza primjerima u Coqu i u dokaznom sustavu $CLKID^{\omega}$. Cijeli rad prožet je ilustrativnim primjerima, kako neformalnih definicija i iskaza, tako i njihovih formalnih reprezentacija u Coqu.

2. Coq

U ovom poglavlju dajemo pregled visoke razine programskog sustava 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 1984. godine [6]. Ovaj rad koristi inačicu 8.18 iz rujna 2023. godine.

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 provjeru tipova 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]

Kao programski jezik, Coq se sastoji od više podjezika različitih namjena, od kojih

[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*.

spominjemo *Vernacular*, *Gallinu* i *Ltac*.

Vernacular („govorni jezik") je jezik naredbi kojima korisnik komunicira sa sustavom (i u interaktivnom i u skupnom načinu rada); svaka Coqova skripta (datoteka s nastavkom `.v`) je niz naredbi. Često korištene naredbe 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 primjer za ilustraciju odnosa ta tri jezika.

```
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] koji je 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 još su istaknuti 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 termu Coq ne može dodijeliti tip. Dalje u radu pod „term" mislimo na dobro formirane terme, odnosno na one kojima Coq 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` ((*pattern matching*)), 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 sprezi 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} : 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 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. Standardni 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 (S n)).
(* Definition from := cofix from (n : nat) := Cons _ n (from (S n)) *)
```

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`).

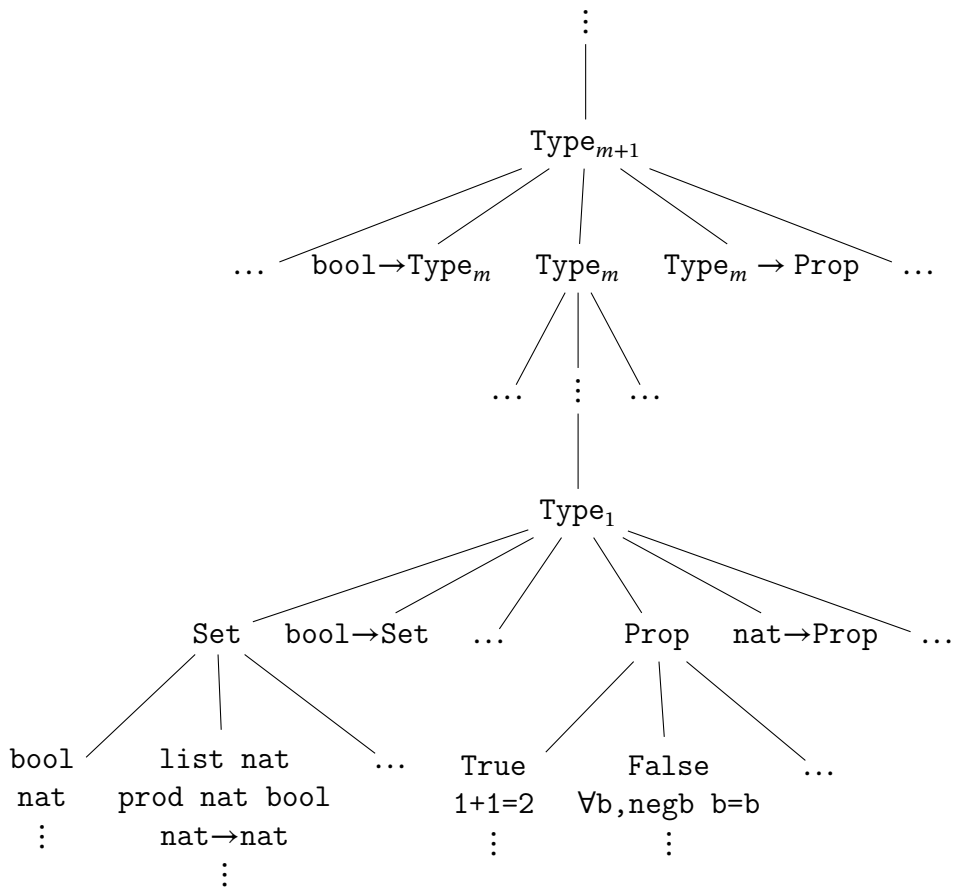
Spomenuli smo da su i tipovi termi te im se može dodijeliti sorta. Postoji li najveća sorta, odnosno postoji li tip `Type` čiji su stanovnici svi tipovi? Prisjetimo se, svaki term ima svoj tip. Kada bi takav `Type` postojao, tada bi vrijedilo `Type : Type`, što može dovesti do paradoksa samoreferenciranja.[11] 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 njene stanovnike zovemo **malim tipovima**. 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**.

Za dokazivače teorema, poželjna je mogućnost definicije predikata (propozicija) nad proizvoljnim tipovima. Zbog toga u Coqu prilikom definicije terma sorte `Prop` možemo

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



Slika 2.1. Kumulativna hijerarhija tipova

kvantificirati po proizvoljno velikim tipovima, što uključuje i sortu `Prop`. Tako je ovdje `not` predikat nad sortom `Prop`.

```
1 Definition not (P : Prop) := forall (Q : Prop), P -> Q.
```

Valja primijetiti da kod definicije predikata `not` kvantificiramo po `Prop`, no za svaku propoziciju `P` je term `not P` također tipa `Prop`, pa domena funkcije `not P` uključuje sam term `not P`. Ovaj stil kvantifikacije omogućuje nam definiciju proizvoljnih propozicijskih veznika.[12] Kažemo da je sorta `Prop` **nepredikativna**. S druge strane, sorta `Set` je **predikativna**, to jest *ne dopušta* kvantifikaciju po `Set` i ostalim proizvoljno velikim tipovima. Kada bi sorta `Prop` bila predikativna, ne bismo mogli definirati predikat `not: Prop → Prop`.

[12] U praksi su veznici induktivno definirani.

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. Ako je pak $n = n' + 1$ za neki n', tada rekurzivno konstruiramo dokaz za $P(n')$, a tražena tvrdnja slijedi primjenom *koraka* indukcije na rekurzivno konstruirani dokaz.

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

```
Definition nat_ind (P : nat -> Prop)
                   (baza : P 0)
                   (korak : forall m, P m -> P (S m))
  : forall n, P n :=
  fix F (n : nat) : P n :=
    match n with
    | O => baza
    | S m => korak m (F m)
    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.[13] 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).[14] 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 |}.
```

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*. Matemati-

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

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

čarima 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: 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 dio adekvatnosti 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 svojom formalizacijom 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 u repozitoriju rada.[1] Zainteresiranom čitatelju predlažemo interaktivni prolazak kroz dokaze lema u nekoj od razvojih okolina za Coq.

Prije no što krenemo na formalizaciju, valja prokomentirati odnos matematičkog i Coqovog vokabulara što se tiče riječi „skup". 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 domene diskursa. 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 prirodnih brojeva E je podskup skupa $\mathbb{N}$, a može se interpretirati kao predikat parnosti na prirodnim brojevima

[1] `https://github.com/mihohren/diploma-thesis`

te možemo pisati $E(n)$ umjesto $n \in E$. U Coqu se skupovi kao 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 prvog reda s induktivnim predikatima* (kratko: *signatura*), s oznakom Σ, je skup simbola od kojih razlikujemo *funkcijske*, *obične predikatne* i *induktivne predikatne* simbole. *Mjesnost* simbola reprezentiramo funkcijom iz odgovarajućeg skupa simbola u skup $\mathbb{N}$, a označujemo ju s $|f|$ za funkcijske, odnosno s $|P|$ za predikatne simbole.

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

Primjer 1. Funkcijski simboli Peanove aritmetike su redom simbol nule o (mjesnosti 0), simbol sljedbenika s (mjesnosti 1) te simboli zbroja $+$ i umnoška $\cdot$ (mjesnosti 2).

```
Inductive Func__PA :=
| PA_zero
| PA_succ
| PA_add
| PA_mult.
Definition fun_ar__PA (s : Func__PA) : nat :=
  match s with
  | PA_zero => 0
  | PA_succ => 1
  | PA_add  => 2
  | PA_mult => 2
  end.
```

[2]Ili tipovima sorte `Type` kada nam trebaju veliki tipovi, najčešće kao posljedica parametrizacije općenitih tipova.

Simbol jednakosti = (mjesnosti 2) je jedini obični predikatni simbol Peanove aritmetike.

```
Inductive Pred__PA := PA_eq.
Definition pred_ar__PA (s : Pred__PA) : nat := 2.
```

Definiramo induktivne predikatne simbole *Nat*, *Even* i *Odd* mjesnosti 1 koje ćemo interpretirati redom kao predikate „biti prirodan", „biti paran" te „biti neparan".

```
Inductive IndPred__PA :=
| PA_Nat
| PA_Even
| PA_Odd.
Definition indpred_ar__PA (s : IndPred__PA) : nat := 1.
```

Konačno, *proširena Peanova signatura*, s oznakom Σ_{PA}, je signatura za Peanovu aritmetiku prvog reda s prethodno definiranim induktivnim predikatnim simbolima.

```
Definition Σ__PA : signature
  := {|
    FuncS := Func__PA;
    fun_ar := fun_ar__PA;
    PredS := Pred__PA;
    pred_ar := pred_ar__PA;
    IndPredS := IndPred__PA;
    indpred_ar := indpred_ar__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 u kasnijim definicijama i iskazima. Coq omogućuje fiksiranje naredbom `Context`, pod uvjetom da se korisnik nalazi u okolini `Section`.[3] Većina definicija i lema u ovom radu su napisane upravo unutar takvih okolina.

Definicija 2. *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 $f(t_1, \ldots, t_n)$ 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 Coqovoj 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.
```

Uobičajene prezentacije logike prvog reda za skup varijabli uzimaju proizvoljni skup $\mathcal{V}$, no za formalizaciju je pogodniji skup prirodnih brojeva $\mathbb{N}$. Umjesto eksplicitne kvantifikacije po nekoj varijabli v, implicitno ćemo kvantificirati po varijabli 0. Ovaj pristup kvantifikaciji[5], nazvan „de Bruijnovo indeksiranje" [15], bitno olakšava rad sa supstitucijama. 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*[6] [17, 18] za automatsko generiranje tipova terma i formula te pripadajućih funkcija supstitucija i pomoćnih lema.

Princip indukcije za `term` potrebno je ručno definirati. Naime, induktivni tip `term` je *ugniježđen* po konstruktoru `TFunc` zato što se javlja omotan oko drugog induktivnog tipa[7] kao argument. Za ugniježđene induktivne tipove, Coq generira neprikladne principe indukcije jer ne zna izraziti tvrdnju „predikat vrijedi za sve ugniježđene elemente".

```
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 terma* t, s oznakom $TV(t)$, konstruiramo rekurzivno na način:

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

```
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. *Skup svih formula* konstruiramo rekurzivno na način:

[5] ili općenitije, vezivanju varijabli
[6] `https://github.com/uds-psl/autosubst2`
[7] ovdje `vec`

1. ako je Q (obični ili induktivni) 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.
```

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)).
```

Primjer 2. U proširenoj Peanovoj signaturi, svojstvo „svaki prirodan broj je paran ili neparan" možemo izraziti formulom $\forall x, Nat(x) \to Even(x) \vee Odd(x)$.

```
Definition every_nat_is_even_or_odd : formula Σ__PA :=
  FAll
    (FImp
       (FIndPred PA_Nat [var_term 0])
       (FOr
          (FIndPred PA_Even [var_term 0])
          (FIndPred PA_Odd  [var_term 0]))).
```

Ovdje vidimo učinak de Bruijnovog indeksiranja na zapis formula; umjesto eksplicitnog navođenja varijable x kod kvantifikacije, implicitno kvantificiramo po varijabli 0.

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

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

```
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* je svaka funkcija iz skupa $\mathbb{N}$ u skup svih terma. Supstituciju σ možemo promatrati kao niz terma $t_0, t_1, t_2, \ldots$. Tada je *pomaknuta supstitucija*, s oznakom $t \cdot \sigma$, supstitucija koja odgovara nizu $t, t_0, t_1, t_2, \ldots$, za neki term t.

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. Često korištene supstitucije formula su supstitucija varijable x termom t u formuli φ, s oznakom $\varphi[t/x]$, te supstitucija svake varijable n u formuli φ varijablom $n+1$, s oznakom $\varphi^{\uparrow}$. 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:

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

Ovdje su $Q_1, \ldots, Q_n$ obični predikatni simboli, $P_1, \ldots, P_m$ i P su induktivni predikatni simboli, a podebljani znakovi predstavljaju n-torke terma, gdje je n mjesnost odgovarajućeg predikata.

Definicija 7. Produkcija je uređena četvorka

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

```
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);
  }.
```

Prvi i drugi član četvorke zovemo *premisama*, a treći i četvrti *konkluzijom*. U ostatku rada odabiremo neki podskup skupa svih produkcija koji zovemo *skupom induktivnih definicija*, a označavamo s Φ.

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

U idućim primjerima definiramo produkcije za induktivne predikatne simbole u proširenoj Peanovoj signaturi.

Primjer 3. Za broj kažemo da je prirodan ako je 0 ili je sljedbenik nekog prirodnog broja.

$$\frac{}{Nat(o)}\,(\texttt{PA\_prod\_N\_zero})$$

```
Definition PA_prod_N_zero : @production Σ__PA.
  refine (mkProd nil nil PA_Nat _).
  refine [TFunc PA_zero []].
Defined.
```

$$\frac{Nat(x)}{Nat(s(x))}\,(\texttt{PA\_prod\_N\_succ})$$

```
Definition PA_prod_N_succ : @production Σ__PA.
  refine (mkProd nil _ PA_Nat _).
  - refine (cons _ nil). exists PA_Nat; refine [var_term 0].
  - refine [TFunc PA_succ [var_term 0]].
Defined.
```

Napomena. U primjeru 3 vidimo alternativni način definiranja u Coqu. Korisnik umjesto definiranja Gallininih terma ulazi u način dokazivanja te naredbom `refine` postepeno konstruira („profinjuje") rezultatni term, ostavljajući „rupe" _ koje će popuniti kasnijim primjenama naredbe `refine`. Konačni rezultat možemo vidjeti naredbom `Print`.

```
Print PA_prod_N_succ.
```

```
PA_prod_N_succ =
{|
  preds := Datatypes.nil;
  indpreds := (PA_Nat; [var_term 0]) :: Datatypes.nil;
  indcons := PA_Nat;
  indargs := [TFunc PA_succ [var_term 0]]
|}
     : production
```

Primjer 4. Za broj kažemo da je paran ako je 0 ili je sljedbenik nekog neparnog broja.

$$\frac{}{Even(o)}\,(\texttt{PA\_prod\_E\_zero})$$

```
Definition PA_prod_E_zero : @production Σ__PA.
  refine (mkProd nil nil PA_Even _).
  refine [ TFunc PA_zero []].
Defined.
```

$$\frac{Odd(x)}{Even(s(x))}\,(\texttt{PA\_prod\_E\_succ})$$

```
Definition PA_prod_E_succ : @production Σ__PA.
  refine (mkProd nil _ PA_Even _).
  - refine (cons _ nil). exists PA_Odd; refine [var_term 0].
  - refine [TFunc PA_succ [var_term 0]].
Defined.
```

Za broj kažemo da je neparan ako je sljedbenik nekog parnog broja.

$$\frac{Even(x)}{Odd(s(x))}\,(\texttt{PA\_prod\_O\_succ})$$

```
Definition PA_prod_O_succ : @production Σ__PA.
  refine (mkProd nil _ PA_Odd _).
  - refine (cons _ nil). exists PA_Even; refine [var_term 0].
  - refine [TFunc PA_succ [var_term 0]].
Defined.
```

Skup prethodnih pet produkcija nazivamo skupom induktivnih definicija za proširenu Peanovu signaturu, s oznakom Φ_{PA}.

```
Inductive Φ__PA : @production Σ__PA -> Prop :=
| ID_N_zero : Φ__PA PA_prod_N_zero
| ID_N_succ : Φ__PA PA_prod_N_succ
| ID_E_zero : Φ__PA PA_prod_E_zero
| ID_E_succ : Φ__PA PA_prod_E_succ
| ID_O_succ : Φ__PA PA_prod_O_succ.
```

3.2. Semantika

Definicija 8. *Struktura prvog reda* (kratko: *struktura*) je uređena četvorka skupa M koji nazivamo *nosačem* te interpretacijā funkcijskih, običnih predikatnih i induktivnih predikatnih simbola. Funkcijski simboli mjesnosti n interpretiraju se kao n-mjesne funkcije, a predikatni simboli mjesnosti n kao n-mjesne relacije na nosaču. Koristit ćemo ime nosača kao sinonim za čitavu strukturu, a interpretacije označavati s 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;
  }.
```

Primjer 5. Prije no što možemo definirati strukturu za Peanovu aritmetiku s induktivnim predikatima, odnosno strukturu za signaturu Σ_{PA} i skup induktivnih definicija Φ_{PA}, potrebno je definirati odgovarajuće interpretacije induktivnih predikatnih simbola u Coqu, u skladu s primjerima 3 i 4.

```
Inductive NAT : nat -> Prop :=
| NO : NAT 0
| NS : forall n, NAT n -> NAT (S n).

Inductive EVEN : nat -> Prop :=
| EO : EVEN 0
| ES : forall n, ODD n -> EVEN (S n)
with ODD : nat -> Prop :=
| OS : forall n, EVEN n -> ODD (S n).
```

Ovdje su predikati `EVEN` i `ODD` definirani mehanizmom simultane rekurzije.

Konačno možemo definirati strukturu M_{PA} s nosačem $\mathbb{N}$ te uobičajenom interpretacijom funkcijskih i običnih predikatnih simbola. Induktivne predikatne simbole Nat, $Even$ i Odd interpretiramo redom Coqovim predikatima `NAT`, `EVEN` i `ODD`.

```
Definition M__PA : @structure Σ__PA.
  refine (Build_structure nat _ _ _).
  - intros f; destruct f.
    + intros. exact 0.
    + intros n. exact (S (V.hd n)).
    + intros xy. exact (V.hd xy + V.hd (V.tl xy)).
    + intros xy. exact (V.hd xy * V.hd (V.tl xy)).
  - intros P args; destruct P.
    exact (V.hd args = V.hd (V.tl args)).
  - intros P args; destruct P.
    + exact (NAT (V.hd args)).
    + exact (EVEN (V.hd args)).
    + exact (ODD (V.hd args)).
Defined.
```

Napomena. U primjeru 5 koristimo taktiku `exact` koja je specijalizacija taktike `refine`, a koristimo ju u zadnjem koraku konstrukcije terma. Nadalje, taktika `intros x` služi uvođenju imena `x` u kontekst, a u našem je slučaju ekvivalentna taktici `refine (fun x => _)`. Konačno, taktika `destruct` služi analizi po slučajevima, a u pozadini koristi *pattern matching*. Kako su interpretacije funkcijskih simbola funkcije koje primaju vektore, koristimo Coqove funkcije `V.hd` i `V.tl` za dohvaćanje glave, odnosno repa vektora.

Definicija 9. Neka je M proizvoljna struktura. *Okolina* ρ za M je proizvoljna funkcija iz skupa varijabli (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*, s oznakom $d \cdot \rho$, 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. Neka je M proizvoljna struktura te ρ okolina za M. *Istinitost* formule φ u okolini ρ, s oznakom $\rho \vDash \varphi$, definiramo rekurzivno na način:

1. ako je P (obični ili induktivni) 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 $P^M(u_1^\rho, \ldots, u_n^\rho)$,
2. vrijedi $\rho \vDash \neg\varphi$ ako i samo ako ne vrijedi $\rho \vDash \varphi$ (što još pišemo $\rho \nvDash \varphi$),
3. vrijedi $\rho \vDash \varphi \to \psi$ ako i samo ako vrijedi $\rho \nvDash \varphi$ ili $\rho \vDash \psi$,
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.
```

Primjer 6. Formula $\forall x, Nat(x) \to Even(x) \vee Odd(x)$ je istinita u strukturi M_{PA} bez obzira na njenu okolinu.

```
Lemma every_nat_is_even_or_odd_Sat :
  forall (ρ : env M__PA), ρ ⊨ every_nat_is_even_or_odd.
```

Lema 1. Neka su φ, σ, M i ρ redom proizvoljna formula, supstitucija, struktura i okolina za M. Tada vrijedi $\rho \vDash \varphi[\sigma]$ ako i samo ako vrijedi $(t \mapsto t^\rho) \circ \sigma \vDash \varphi$.

```
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.
```

Kompoziciju supstitucije σ i evaluacije $t \mapsto t^{\rho}$ možemo nazvati *semantičkom* supstitucijom jer prvo provodi *sintaksnu* supstituciju σ nakon čega provodi evaluaciju. Tada možemo neformalno reći da sintaksna i semantička supstitucija komutiraju pod relacijom istinitosti.

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 najviše jednom primjenom produkcija, druga je razina onaj podskup kojeg možemo dobiti pomoću najviše dviju primjena produkcija, i tako dalje. Na taj se način, korak po korak, gradi *smislena* interpretacija induktivnih predikata. Zbog mogućih međuovisnosti induktivnih predikata, razine interpretacije moramo definirati simultano. Ovaj odjeljak posvećujemo formalizaciji ovih pojmova.

Definicija 11. Neka je M proizvoljna struktura te neka je pr proizvoljna produkcija induktivnog skupa definicija Φ, 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 I proizvoljna interpretacija induktivnih predikatnih simbola. Tada definiramo $\varphi_{pr}(I)$ kao 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 $\mathbf{v}_j^{\rho} \in f(P_j)$ i
- $\mathbf{d} = \mathbf{t}^{\rho}$.

```
Definition φ_pr
  (pr : production Σ)
  (interp : InterpInd) (* interpretacija I *)
  (ds : vec M (indpred_ar (indcons pr)))
  : Prop :=
      exists (ρ : env M),
      (forall Q us, List.In (Q; us) (preds pr) ->
                interpP Q (V.map (eval ρ) us)) /\
        (forall P ts, List.In (P; ts) (indpreds pr) ->
                  interp P (V.map (eval ρ) ts)) /\
        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 konkluziji.

```
Definition φ_P
  (P : IndPredS Σ)
  (interp : InterpInd)
  : vec M (indpred_ar P) -> Prop.
  refine (fun ds => _).
  refine (@ex (production Σ) (fun pr => _)).
  refine (@ex (P = indcons pr /\ Φ pr) (fun '(conj Heq HΦ) => _)).
  rewrite Heq in ds.
  exact (φ_pr pr interp ds).
Defined.
```

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

```
Definition φ_Φ (interp : InterpInd) : InterpInd :=
  fun P => φ_P P interp.
```

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 $\varphi_\Phi(f)(P) \coloneqq \varphi_P(f)$.

Primjer 7. Ilustrirajmo operator φ_Φ na strukturi M_{PA} i skupu induktivnih definicija Φ_{PA}. Prisjetimo se, u proširenoj Peanovoj signaturi imamo tri induktivna predikatna simbola; Nat, $Even$ i Odd, i svi su oni jednomjesni. Dakle, domena i kodomena operatora φ_Φ u ovom primjeru je skup $\mathcal{P}(\mathbb{N})^3$, gdje prva, druga i treća projekcija odgovaraju redom

interpretacijama predikata Nat, $Even$ i Odd. Nadalje, vrijedi:

$$\varphi_{\Phi_{PA},M_{PA}}(N,E,O) = (\{0\} \cup \{n+1 \mid n \in N\}, \{0\} \cup \{o+1 \mid o \in O\}, \{e+1 \mid e \in E\}).$$

Prema notaciji u prethodnoj napomeni, funkcija f je ovdje preslikavanje

$$(Nat \mapsto N, Even \mapsto E, Odd \mapsto O)$$

koje možemo reprezentirati uređenom trojkom (N,E,O). Indeksiranje induktivnim predikatnim simbolom odgovara prethodno navedenim projekcijama pa vrijedi primjerice:

$$\varphi_{\Phi_{PA},M_{PA}}(N,E,O)(Nat) = \{0\} \cup \{n+1 \mid n \in N\}.$$

Definicija 12. Neka su U i V proizvoljne interpretacije induktivnih predikatnih simbola. Kažemo da je interpretacija U *podinterpretacija* interpretacije V i pišemo $U \sqsubseteq V$ ako za svaki induktivni predikatni simbol P vrijedi $U(P) \subseteq V(P)$.

```
Definition subinterp (U V : InterpInd) := forall P v, U P v -> V P v.
Notation "U ⊑ V" := (subinterp U V) (at level 11).
```

Propozicija 1. Operator φ_Φ je monoton.

```
Definition monotone (F : InterpInd -> InterpInd) :=
  forall U V : InterpInd, U ⊑ V -> F U ⊑ F V.

Proposition φ_Φ_monotone : monotone φ_Φ.
```

Definicija 13. Neka je M proizvoljna struktura. Definiramo *aproksimaciju* skupa induktivnih definicija Φ razine $\alpha \in \mathbb{N}$, s oznakom φ_Φ^α, rekurzivno na način:

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

Neformalno možemo reći da operator φ_Φ *poboljšava* aproksimaciju prethodne razine.

```
Fixpoint φ_Φ_n (α : nat) : InterpInd :=
  match α with
  | 0 => fun _ _ => False
  | S α => φ_Φ (φ_Φ_n α)
  end.
```

Tada je *aproksimant* induktivnog predikatnog simbola P razine α upravo $\varphi_\Phi^\alpha(P)$.

Napomena. Brotherston je definirao aproksimaciju razine α za pojedini induktivni predikatni simbol kao uniju poboljšanih aproksimacija svih nižih razina. Takva je definicija ekvivalentna našoj.

Lema 2. Za svaki prirodni broj α i induktivni predikatni simbol P vrijedi

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

```
Lemma φ_Φ_n_characterization : forall α P v,
    φ_Φ_n α P v <-> exists β, β < α /\ φ_Φ (φ_Φ_n β) P v.
```

Primjer 8. Za svaki prirodni broj α, a u skladu s primjerom 7, aproksimant razine α induktivnog predikatnog simbola Nat je skup svih prirodnih brojeva strogo manjih od α, odnosno vrijedi:

$$\varphi^\alpha_{\Phi_{PA},M_{PA}}(Nat) = \{0, 1, \ldots, \alpha - 1\}.$$

```
Lemma approximants_of_PA_Nat : forall α n,
    φ_Φ_n Φ__PA M__PA α PA_Nat [n] <-> n < α.
```

Definicija 14. *Aproksimacija razine* ω, s oznakom φ_Φ^ω, je (za svaki pojedini predikatni simbol) unija aproksimacija razina manjih od ω.

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

Primjer 9. Aproksimant razine ω induktivnog predikatnog simbola Nat je cijeli skup prirodnih brojeva $\mathbb{N}$.

```
Lemma NAT_φ_Φ_ω: forall n, φ_Φ_ω Φ__PA M__PA PA_Nat [n].
```

Definicija 15. Neka je U proizvoljna interpretacija induktivnih predikatnih simbola. Kažemo da je interpretacija U *prefiksna točka* ako je poboljšanje interpretacije U podinterpretacija od U, odnosno vrijedi $\varphi_\Phi(U) \sqsubseteq U$. Za interpretaciju induktivnih predikatnih simbola V kažemo da je *najmanja prefiksna točka* ako je V prefiksna točka te je podinterpretacija svake druge prefiksne točke.

```
Definition prefixed (U : InterpInd) := φ_Φ U ⊑ U.
Definition least (P : InterpInd -> Prop) (U : InterpInd) :=
  P U /\ (forall V, P V -> U ⊑ V).
```

Lema 3. Aproksimacija razine ω je najmanja prefiksna točka operatora φ_Φ.

```
Lemma φ_Φ_ω_least_prefixed : least prefixed φ_Φ_ω.
```

Definicija 16. Kažemo da je struktura M *standardni model* za Φ ako interpretira svaki induktivni predikatni simbol njegovim aproksimantom razine ω.

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

Prema lemi 3, standardni model je najmanja struktura koja ima smisla za zadani skup induktivnih definicija.

Propozicija 2. Struktura M_{PA} je standardni model za skup induktivnih definicija Φ_{PA}.

```
Lemma standard_model__PA : standard_model Φ__PA M__PA..
```

4. Sistem sekvenata s induktivnim definicijama

Cilj je ovog odjeljka definirati dokazni sustav *LKID* za logiku FOL_{ID}. Ovaj dokazni sustav temelji se na Gentzenovu računu sekvenata, a proširen je lijevim i desnim pravilima za induktivne predikate. Prije no što definiramo *LKID*, potrebno je definirati pojam sekvente i međusobne zavisnosti induktivnih predikata.

Definicija 17. Neka su Γ i Δ proizvoljne liste formula. Tada je *sekventa* uređeni par (Γ, Δ), a označavamo ju s $\Gamma \vdash \Delta$.

```
Inductive sequent : Set :=
| mkSeq (Γ Δ : list (formula Σ)).
```

Sekventom $\Gamma \vdash \Delta$ konceptualno tvrdimo da istinitost *svake* tvrdnje u Γ povlači istinitost *neke* tvrdnje u Δ. Kažemo da je sekventa $\Gamma \vdash \Delta$ *dokaziva*, ili da Γ *izvodi* Δ, ako u sustavu *LKID* postoji *dokaz* za $\Gamma \vdash \Delta$. Neformalno možemo reći da je dokaz sekvente $\Gamma \vdash \Delta$ konačno stablo u kojem je sekventa $\Gamma \vdash \Delta$ korijen te svaki čvor stabla slijedi iz svoje djece u skladu s pravilima izvoda sustava *LKID*. Kasnije ćemo precizirati pojam dokaza.

Definicija 18. 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 njenoj konkluziji 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 19. Definiramo relaciju $Prem^*$ kao refleksivno i tranzitivno zatvorenje relacije *Prem* na skupu induktivnih predikatnih simbola.

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

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.
```

Za međusobno zavisne induktivne predikatne simbole kažemo još da su definirani simultanom rekurzijom. Simultana rekurzija se koristi za formalnu definiciju međusobno zavisnih induktivnih predikata i u mnogo složenijim sustavima od FOL_{ID}; primjerice, Coqovi predikati `EVEN` i `ODD` iz primjera 5 definirani su simultanom rekurzijom.

Primjer 10. U skupu induktivnih definicija Φ_{PA}, induktivni predikatni simboli *Even* i *Odd* su međusobno zavisni.

```
Example mut_dep_E_O :
  mutually_dependent Φ__PA PA_Even PA_Odd.
```

S druge strane, induktivni predikatni simboli *Nat* i *Even nisu* međusobno zavisni.

```
Example not_mut_dep_N_E :
  ~mutually_dependent Φ__PA PA_Nat PA_Even.
```

Lema 4. Međusobna zavisnost je relacija ekvivalencije na skupu induktivnih predikatnih simbola.

```
Lemma mutually_dependent_equiv : Equivalence mutually_dependent.
```

Sustav *LKID* sastoji se od četiri vrste pravila izvoda: strukturalna, propozicijska, pravila za kvantifikatore i pravila za induktivne predikate. Navest ćemo ih tim redom, a definirati kroz više blokova Coqova koda. Napominjemo da se sustav *LKID* razlikuje od Gentzenova sustava *LK* na nekoliko mjesta, zbog implementacijskih detalja te zbog induktivnih definicija. Za neko pravilo izvoda kažemo da je *dopustivo* ako sve što možemo dokazati (pod nekim pretpostavkama) korištenjem tog pravila, možemo dokazati pod istim pretpostavkama i bez njegova korištenja.

```
Inductive LKID : sequent -> Prop := (* nastavlja se *)
```

4.1. Strukturalna pravila

Strukturalna su pravila prikazana na slici 4.1., a služe baratanju strukturom sekvente. Iako je sekventa implementirana kao par lista formula, te liste se ponašaju kao skupovi u strukturalnim pravilima jer za sadržanost jedne liste u drugoj koristimo Coqov predikat `incl` koji se oslanja na predikat `In`. Posljedično, pravilo slabljenja Wk povlači pravila permutacije i kontrakcije.

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

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

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

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

Slika 4.1. Strukturalna pravila sustava *LKID*.

```
(* Strukturalna pravila. *)
| 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)
```

Primjer 11. Prošireno pravilo aksioma je dopustivo.

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

```
Lemma AxExtended : forall Γ Δ φ, LKID (φ :: Γ ⊢ φ :: Δ).
```

Nadalje, pravilo permutacije je dopustivo.

$$\frac{\Gamma' \vdash \Delta' \qquad \Gamma' \sim \Gamma \qquad \Delta' \sim \Delta}{\Gamma \vdash \Delta}\,(Perm)$$

Ovdje znakom $\sim$ označujemo relaciju permutacije na listama.

```
Lemma Perm : forall Γ' Δ' Γ Δ,
    Permutation Γ' Γ ->
    Permutation Δ' Δ ->
    LKID (Γ' ⊢ Δ') ->
    LKID (Γ ⊢ Δ).
```

Konačno, lijeva i desna pravila kontrakcije su dopustiva.

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

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

```
Lemma ContrL : forall Γ Δ φ,
    LKID (φ :: φ :: Γ ⊢ Δ) -> LKID (φ :: Γ ⊢ Δ).
Lemma ContrR : forall Γ Δ φ,
    LKID (Γ ⊢ φ :: φ :: Δ) -> LKID (Γ ⊢ φ :: Δ).
```

4.2. Propozicijska pravila

Propozicijska su pravila prikazana na slici 4.2., a služe rasuđivanju s implikacijama i negacijama. Kao što smo rekli, ostale propozicijske veznike shvaćamo kao pokrate. Ova su pravila identična u sustavima *LK* i *LKID*.

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

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

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

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

Slika 4.2. Propozicijska pravila sustava *LKID*.

```
(* Propozicijska pravila. *)
| 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)
```

Primjer 12. Desno pravilo za konjunkciju je dopustivo.

$$\frac{\Gamma \vdash \varphi, \Delta \qquad \Gamma \vdash \psi, \Delta}{\Gamma \vdash \varphi \wedge \psi, \Delta}\,(AndR)$$

```
Lemma AndR : forall Γ Δ φ ψ,
    LKID (Γ ⊢ φ :: Δ) -> LKID (Γ ⊢ ψ :: Δ) ->
    LKID (Γ ⊢ FAnd φ ψ :: Δ).
```

Lijevo pravilo za disjunkciju je dopustivo.

$$\frac{\varphi, \Gamma \vdash \Delta \qquad \psi, \Gamma \vdash \Delta}{\varphi \vee \psi, \Gamma \vdash \Delta}\,(OrL)$$

```
Lemma OrL : forall Γ Δ φ ψ,
    LKID (φ :: Γ ⊢ Δ) -> LKID (ψ :: Γ ⊢ Δ) ->
    LKID (FOr φ ψ :: Γ ⊢ Δ).
```

4.3. Pravila za kvantifikatore

Pravila za kvantifikatore prikazana su na slici 4.3., a odnose se na univerzalnu kvantifikaciju. Ova se pravila razlikuju od analognih u sustavu *LK* zbog korištenja de Bruijnovih indeksa kod kvantifikacije. U lijevom pravilu koristimo supstituciju $t \cdot \sigma_{id}$ u formuli φ, gdje je σ_{id} supstitucija identiteta, kako bismo iskazali da se term t javlja na mjestu neke varijable x. Za desno pravilo, primijetimo da se varijabla 0 ne javlja u listama formula $\Gamma^{\uparrow}$ i $\Delta^{\uparrow}$ zbog pomicanja. Kako implicitno kvantificiramo po varijabli 0, a ona se javlja (eventualno) samo u formuli φ, smijemo univerzalno kvantificirati formulu φ.

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

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

Slika 4.3. Pravila za kvantifikatore sustava *LKID*.

```
(* Kvantifikatorska pravila. *)
| 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)
```

Primjer 13. Lijevo pravilo za egzistencijalnu kvantifikaciju je dopustivo.

$$\frac{\varphi, \Gamma^{\uparrow} \vdash \Delta^{\uparrow}}{\exists\varphi, \Gamma \vdash \Delta}\,(ExistL)$$

```
Lemma ExistL : forall Γ Δ φ,
    LKID (φ :: shift_formulas Γ ⊢ shift_formulas Δ) ->
    LKID (FExist φ :: Γ ⊢ Δ).
```

Preostaje nam definicija pravila za induktivne predikatne simbole, gdje ćemo prvo za svaki induktivni predikatni simbol definirati lijevo pravilo koje nazivamo *pravilo indukcije*, a zatim ćemo za svaku produkciju definirati desno pravilo koje nazivamo *produkcijsko pravilo*.

4.4. Pravila indukcije

Neka je P_j proizvoljni induktivan predikatni simbol za koji definiramo pravilo indukcije. Svakom induktivnom predikatnom simbolu P_i dodjeljujemo proizvoljnu $|P_i|$-torku *induktivnih varijabli* $\mathbf{z}_i$ te proizvoljnu *pretpostavku indukcije* G_i takvu da vrijedi $G_i = P_i(\mathbf{z}_i)$ ako P_i i P_j nisu međusobno zavisni. Sada pravilo indukcije za induktivan predikatni simbol P_j glasi:

$$\frac{\textit{male premise} \qquad G_j[\mathbf{u}/\mathbf{z}_i], \Gamma \vdash \Delta}{P_j\mathbf{u}, \Gamma \vdash \Delta}\,(P_j\,Ind),$$

gdje sekventu $G_j[\mathbf{u}/\mathbf{z}_i], \Gamma \vdash \Delta$ zovemo *velikom premisom*, a za svaku produkciju u skupu induktivnih definicija Φ u čijoj se konkluziji javlja induktivni predikatni simbol P_i međusobno zavisan sa simbolom P_j, primjerice:

$$\frac{Q_1\mathbf{u}_1 \dots Q_n\mathbf{u}_n \qquad P_{i_1}\mathbf{v}_{i_1} \dots P_{i_m}\mathbf{v}_{i_m}}{P_i\mathbf{t}},$$

postoji *mala premisa* oblika:

$$Q_1\mathbf{u}_1^{\uparrow k}, \dots, Q_n\mathbf{u}_n^{\uparrow k}, G_{i_1}[\mathbf{v}_{i_1}^{\uparrow k}/\mathbf{z}_{i_1}], \dots, G_{i_m}[\mathbf{v}_{i_m}^{\uparrow k}/\mathbf{z}_{i_m}], \Gamma \vdash G_i[\mathbf{t}^{\uparrow k}/\mathbf{z}_i], \Delta$$

gdje je k neki prirodan broj takav da se nijedan $k' \geq k$ ne pojavljuje u listama formula Γ i Δ te formuli $P_j\mathbf{u}$ (takav postoji jer se u tih konačno mnogo formula sveukupno pojavljuje samo konačno mnogo varijabli), a s $^{\uparrow k}$ označavamo supstituciju koja svaku varijablu n zamjenjuje varijablom $n + k$.

Ilustrirajmo pravilo indukcije primjerima u Peanovoj aritmetici. Kako se pravila za induktivne predikate sustava *LKID* odnose na proizvoljni skup induktivnih definicija Φ, označimo s $LKID_{PA}$ sustav koji se odnosi na skup induktivnih defincija Φ_{PA}.

```
1 Definition LKID__PA := LKID Φ__PA.
```

Primjer 14. Pravilo indukcije simbola Nat u sustavu $LKID_{PA}$ glasi:

$$\frac{\Gamma \vdash G(o), \Delta \qquad G(x), \Gamma \vdash G(s(x)), \Delta \qquad G(t), \Gamma \vdash \Delta}{Nat(t), \Gamma \vdash \Delta}\,(NatInd),$$

gdje je G pretpostavka indukcije vezana uz induktivni predikatni simbol Nat, a x je varijabla koja se ne javlja u Γ, Δ niti u $Nat(t)$.

Primjer 15. Pravilo indukcije simbola $Even$ u sustavu $LKID_{PA}$ glasi:

$$\frac{\Gamma \vdash G_E(o), \Delta \quad \Gamma, G_E(x) \vdash G_O(s(x)), \Delta \quad \Gamma, G_O(x) \vdash G_E(s(x)), \Delta \qquad \Gamma, G_E(t) \vdash \Delta}{\Gamma, Even(t) \vdash \Delta}\,(EvenInd),$$

gdje su G_E i G_O pretpostavke indukcije vezane uz induktivne predikatne simbole $Even$ i Odd, a x je varijabla koja se ne javlja u Γ, Δ niti u $Even(t)$.

```
(* Pravila indukcije. *)
| IndL : forall Γ Δ
           (Pj : IndPredS Σ) (u : vec _ (indpred_ar Pj))
           (z_i : forall P, vec var (indpred_ar P))
           (z_i_nodup : forall P, VecNoDup (z_i P))
           (G_i : IndPredS Σ -> formula Σ)
           (HG2 : forall Pi, ~mutually_dependent Pi Pj ->
                        G_i Pi = FIndPred
                                   Pi
                                   (V.map var_term (z_i 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 (HΦ:Φ pr) (Hdep: mutually_dependent (indcons pr) Pj),
          let Qs := shift_formulas_by
                      shift_factor
                      (FPreds_from_preds (preds pr)) in
          let Gs := map (fun '(P; args) =>
                           let shifted_args :=
                             V.map
                               (shift_term_by shift_factor)
                               args in
                           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))
    in
    minor_premises ->
    LKID (Fj :: Γ ⊢ Δ) ->
    LKID (FIndPred Pj u :: Γ ⊢ Δ)
```

4.5. Produkcijska pravila

Neka je *pr* proizvoljna produkcija za koju definiramo produkcijsko pravilo. Možemo pretpostaviti da je *pr* oblika:

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

Tada produkcijsko pravilo za produkciju *pr* (i proizvoljnu supstituciju σ) glasi:

$$\frac{\Gamma \vdash Q_1\mathbf{u}_1[\sigma], \Delta \quad \ldots \quad \Gamma \vdash Q_n\mathbf{u}_n[\sigma] \qquad \Gamma \vdash P_1\mathbf{v}_1[\sigma], \Delta \quad \ldots \quad \Gamma \vdash P_m\mathbf{v}_m[\sigma]}{\Gamma \vdash P\mathbf{t}[\sigma]}\ (prProd).$$

```
58 (* Produkcijska pravila. *)
59 | IndR : forall Γ Δ pr σ,
60     Φ pr ->
61     (forall Q us,
62         In (Q; us) (preds pr) ->
63         LKID (Γ ⊢ (FPred Q (V.map (subst_term σ) us) :: Δ))) ->
64     (forall P ts,
65         In (P; ts) (indpreds pr) ->
66         LKID (Γ ⊢ (FIndPred P (V.map (subst_term σ) ts) :: Δ))) ->
67     LKID ( Γ ⊢ FIndPred
68               (indcons pr)
69               (V.map (subst_term σ) (indargs pr))
70               :: Δ).
```

Primjer 16. Produkcijsko pravilo za produkciju `PA_prod_E_succ` glasi:

$$\frac{\Gamma \vdash Odd(x)[\sigma], \Delta}{\Gamma \vdash Even(s(x))[\sigma], \Delta}\ (\texttt{PA\_prod\_E\_succ}Prod)$$

gdje je x proizvoljna varijabla, a σ proizvoljna supstitucija. Supstitucijom σ kažemo da u produkcijama imena varijabli nisu bitna, već samo njihov redoslijed.

4.6. Dokaz i teorem

Prikazali smo sva pravila izvoda sustava *LKID* te konačno možemo definirati dokaz. Za proizvoljnu sekventu $s := (\Gamma \vdash \Delta)$, njen **dokaz** je stanovnik tipa `LKID s`. Za formulu φ kažemo da je **dokaziva** u sustavu *LKID* te pišemo $\vdash \varphi$ ako postoji dokaz sekvente $[] \vdash [\varphi]$.

```
Definition provable (proofsystem : sequent -> Prop) (φ : formula Σ) :=
  proofsystem ([] ⊢ [φ]).
```

Neformalni pojam konačnog stabla formaliziran je induktivnim tipom `LKID`, dok njegovi konstruktori formaliziraju odgovarajuća pravila izvoda.

Primjer 17. Za proizvoljnu formulu φ, formula $\varphi \vee \neg\varphi$ je dokaziva u sustavu *LKID*.

```
Lemma LKID_XM : forall φ, provable LKID (FOr φ (FNeg φ)).
```

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\varphi \vdash \varphi}(Ax)}{\vdash \neg\varphi, \varphi}(NegR)}{\vdash \varphi, \neg\varphi}(Perm)}{\vdash \varphi \vee \neg\varphi}(OrR)$$

Primjer 18. Za sustav *LKID* vrijedi *princip eksplozije*.

```
Lemma LKID_EXPLOSION : forall φ Δ, LKID ([FAnd φ (FNeg φ)] ⊢ Δ).
```

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\varphi \vdash \varphi, \Delta}(Ax)}{\neg\varphi, \varphi \vdash \Delta}(NegL)}{\varphi, \neg\varphi \vdash \Delta}(Perm)}{\varphi \wedge \neg\varphi \vdash \Delta}(AndL)$$

Primjer 19. Formula $\forall x, Even(x) \to Even(s(s(x)))$ je dokaziva u sustavu $LKID_{PA}$.

```
Definition Even_succ_succ_Even : formula Σ__PA :=
  let x := var_term 0 in
  FAll
    (FImp
       (FIndPred PA_Even [x])
       (FIndPred PA_Even [TFunc PA_succ
                            [TFunc PA_succ
                               [x]]])).
```

Neformalni dokaz prikazujemo idućim stablom, a radi preglednosti pišemo *Et* umjesto $Even(t)$ te *Ot* umjesto $Odd(t)$ i ne pišemo zagrade u pozivu funkcijskog simbola s.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{Ex \vdash Ex, Essx}(Ax)}{Ex \vdash Osx, Essx}(\texttt{PA\_prod\_0\_succ}Prod) \quad \dfrac{\dfrac{}{Ex, Osx \vdash Osx}(Ax)}{Ex, Osx \vdash Essx}(\texttt{PA\_prod\_E\_succ}Prod)}{Ex \vdash Essx}(Cut)}{\vdash Ex \to Essx}(ImpR)}{\vdash \forall x, Ex \to Essx}(AllR)$$

```
Lemma provable_every_succ_of_Even_is_Odd :
  provable LKID__PA every_succ_of_Even_is_Odd.
```

Primjer 20. Formula $\forall x, Nat(x) \to Even(x) \vee Odd(x)$ je dokaziva u sustavu $LKID_{PA}$. Neformalni dokaz prikazujemo idućim stablom koristeći iste pokrate kao u primjeru 19, te Nt za $Nat(t)$.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash Eo}\,(1)}{\vdash Eo, Ex \vee Ox}\,(Wk)}{\vdash Eo \vee Oo, Ex \vee Ox}\,(OrR_1) \qquad \dfrac{\dfrac{\dfrac{}{Ey \vdash Ey}\,(Ax)}{Ey \vdash Osy}\,(2)}{Ey \vdash Esy \vee Osy}\,(OrR_2) \qquad \dfrac{\dfrac{}{Oy \vdash Oy}\,(Ax)}{Oy \vdash Esy}\,(3)}{Oy \vdash Esy \vee Osy}\,(OrR_1)}{Ey \vee Oy \vdash Esy \vee Osy}\,(OrL)}{Ey \vee Oy \vdash Esy \vee Osy, Ex \vee Ox}\,(Wk) \qquad \dfrac{}{Ex \vee Ox \vdash Ex \vee Ox}\,(Ax)}{\dfrac{\dfrac{Nx \vdash Ex \vee Ox}{Nx \to Ex \vee Ox}\,(ImpR)}{\vdash \forall x, Nx \to Ex \vee Ox}\,(AllR)}\,(NatInd)$$

Ovdje je pretpostavka indukcije formula $Ex \vee Ox$. Valja primijetiti da smo u dokazu koristili (označene brojevima) i produkcijska pravila za produkcije:

(1) `PA_prod_E_zero`,

(2) `PA_prod_O_succ` i

(3) `PA_prod_E_succ`.

```
Lemma provable_every_nat_is_even_or_odd :
  provable LKID__PA every_nat_is_even_or_odd.
```

4.7. Adekvatnost

Za dokazivače teorema nužno je da dokazivost formule povlači njenu istinitost — u 2. poglavlju objasnili smo zašto. Ovo svojstvo dokaznog sustava naziva se *adekvatnost*, a ovdje ćemo je prikazati na primjeru sustava *LKID*. Za istinitost ćemo promatrati samo strukture koje su standardni modeli, jer u takvima interpretacije induktivnih predikatnih simbola imaju smisla. Napomenimo da je *LKID* adekvatan i za nestandardne modele.

Definicija 20. Kažemo da je sekventa $\Gamma \vdash \Delta$ *istinita* i pišemo $\Gamma \models \Delta$ ako i samo ako za svaki standardni model M i njegovu okolinu ρ vrijedi:

$$(\forall \varphi, \varphi \in \Gamma \to \rho \vDash \varphi) \to \exists \psi, \psi \in \Delta \wedge \rho \vDash \psi,$$

odnosno istinitost svake formule u Γ povlači istinitost neke formule u Δ, u skladu s intuitivnim shvaćanjem sekventi.

```
Definition Sat_sequent (s : sequent) : Prop :=
  let '(Γ ⊢ Δ) := s in
  forall (M : structure Σ), standard_model Σ Φ M -> forall (ρ : env M),
      (forall φ, In φ Γ -> ρ ⊨ φ) -> exists ψ, In ψ Δ /\ ρ ⊨ ψ.
```

Kažemo da vrijedi *lokalna adekvatnost* za pravilo izvoda ako istinitost (sekventi) premisa povlači istinitost sekvente konkluzije. Lokalna adekvatnost kaže da se istinitost čuva kod primjena pravila zaključivanja.

Primjer 21. Za pravilo Ax vrijedi lokalna adekvatnost, to jest ako u listama formula Γ i Δ postoji neka zajednička formula φ, onda vrijedi $\Gamma \Vdash \Delta$. Tada je *svjedok istinitosti* u Δ upravo formula φ.

```
Lemma LS_Ax : forall Γ Δ φ, In φ Γ -> In φ Δ -> Γ ⊩ Δ.
```

Ako za svako pravilo izvoda vrijedi lokalna adekvatnost, indukcijom po strukturi dokaza slijedi i (globalna) adekvatnost. Glavni cilj ovog poglavlja bio je dokazati lokalne adekvatnosti svih pravila sustava $LKID$. Cilj je postignut za sva pravila osim pravila indukcije.

Teorem 1. Za pravila izvoda sustava $LKID$ osim $IndL$ vrijedi lokalna adekvatnost. Posljedično, ako je sekventa $\Gamma \vdash \Delta$ dokaziva bez primjene pravila $IndL$, onda je ona istinita.

5. Ciklički dokazi

Svaki Coqov dokaz koji smo do sada prikazali je konačan term. Svi formalni dokazi u sustavu *LKID* (uključujući primjere 19 i 20) su konačni. Ovakve dokaze jednim imenom nazivamo *dokazima konačne dubine* jer su sve „grane" njihovih dokaznih stabala konačne. Međutim, možemo promatrati i *dokaze beskonačne dubine* kod kojih postoje beskonačne grane. Već smo u odjeljku 2.2. spomenuli koinduktivne tipove, a dokaz beskonačne dubine je tada „beskonačni" dokaz neke koinduktivno definirane tvrdnje. U ovom ćemo kratkom poglavlju ilustrirati dokaze beskonačne dubine te jednu njihovu posebnu formu: *cikličke* dokaze.

5.1. Primjer: lijene liste

Definirajmo za početak *lijene liste*, koje mogu biti konačne, a mogu biti i beskonačne. Zbog potencijalne beskonačnosti koristimo koinduktivnu definiciju tipa.

```
CoInductive LList (A : Type) :=
| LNil : LList A
| LCons : A -> LList A -> LList A.
```

Kažemo da je lijena lista *beskonačna* ako je njen rep opet beskonačna lijena lista.

```
CoInductive Infinite {A} : LList A -> Prop :=
| Infinite_LCons : forall a l, Infinite l -> Infinite (LCons a l).
```

Ovdje konstruktor `Infinite_LCons` možemo shvatiti kao pravilo izvoda. Pokušajmo dokazati da je neka intuitivno beskonačna lista i formalno beskonačna. Definiramo funkciju `from` analogno istoimenoj funkciji iz odjeljka 2.2.

```
CoFixpoint from (n : nat) : LList nat := LCons n (from (S n)).
```

Intuitivno je jasno da je (za svaki `n`) lijena lista `from n` beskonačna pa želimo dokazati

tvrdnju `forall n, Infinite (from n)`.

Prije glavnog dokaza, pogledajmo konkretan primjer na tvrdnji `Infinite (from 0)`. Po definiciji funkcije `from`, dovoljno je dokazati tvrdnju `Infinite (LCons 0 (from 1))`, a primjenom pravila `Infinite_LCons` dolazimo do tvrdnje `Infinite (from 1)`. Opet je po definiciji funkcije `from` dovoljno dokazati tvrdnju `Infinite (LCons 1 (from 2))`, a ponovnom primjenom pravila `Infinite_LCons` dolazimo do tvrdnje `Infinite (from 2)`. Ilustrirajmo ovaj postupak dokaznim stablom.

$$\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{\texttt{Infinite (from 2)}}}{\texttt{Infinite (LCons 1 (from 2))}}\,(\texttt{Infinite\_LCons})}{\texttt{Infinite (from 1)}}}{\texttt{Infinite (LCons 0 (from 1))}}\,(\texttt{Infinite\_LCons})$$
$$\texttt{Infinite (from 0)}$$

Sada je jasno da ćemo dokazati originalnu tvrdnju tek nakon beskonačno mnogo primjena pravila `Infinite_LCons`. Ipak, lakše je dokazati tvrdnju `forall n, Infinite (from n)`. Nakon introdukcije varijable `n`, odmatanja definicije funkcije `from` i primjene pravila `Infinite_LCons` dolazimo do tvrdnje `Infinite (from (S n))`.

$$\dfrac{\dfrac{\dfrac{\dfrac{*}{\texttt{Infinite (from (S n))}}}{\texttt{Infinite (LCons n (from (S n)))}}\,(\texttt{Infinite\_LCons})}{\texttt{Infinite (from n)}}}{\texttt{forall n, Infinite (from n)}}$$

Između tvrdnji `Infinite (from n)` i `Infinite (from (S n))` „skinut" je prvi član beskonačne liste, čime je napravljen *napredak* u dokazu. Bez napretka nema smisla ponavljati prethodni postupak beskonačno mnogo puta jer se tvrdnja ne mijenja — u suprotnom, mogli bismo svaku tvrdnju φ dokazati trivijalnim dokazom $\frac{*}{\varphi}$. Uz napredak u dokazivanju ima smisla pozivati se na *tvrdnju koja se dokazuje* te takve dokaze nazivamo *cikličkima*. Dokaz tvrdnje `forall n, Infinite (from n)` jedan je primjer cikličkog dokaza, gdje je simbolom $*$ označeno cikličko „pozivanje" dokaza. U Coqu, napredak je osiguran uvjetom produktivnosti spomenutim u odjeljku 2.5.

Pažljiv čitatelj će primijetiti da ista argumentacija vrijedi i za tvrdnju `Finite (from n)`, gdje je predikat `Finite` definiran na idući način.

```
Inductive Finite {A} : LList A -> Prop :=
| Finite_LNil : Finite LNil
| Finite_LCons : forall a l, Finite l -> Finite (LCons a l).
```

Međutim, beskonačne liste po definiciji nisu konačne. Zašto ipak gornji dokazi *ne* prolaze za predikat `Finite`? Riječ je o načinu definicije; predikat `Infinite` definiran je koinduktivno te njegovi stanovnici *nisu* dobro utemeljeni, dok je predikat `Finite` definiran induktivno, a njegovi stanovnici *jesu* dobro utemeljeni. Štoviše, kada bismo predikat `Finite` definirali koinduktivno, on bi postao trivijalan, to jest vrijedio bi za proizvoljnu lijenu listu.

5.2. Primjer: $CLKID^{\omega}$

Brotherston je formalizirao ideju cikličkih dokaza u dokaznom sustavu $CLKID^{\omega}$, podsustavu dokaznog sustava $LKID^{\omega}$, koji je pak proširenje sustava $LKID$ potencijalno beskonačnim dokaznim stablima. Dokazni sustav $CLKID^{\omega}$ ograničen je u odnosu na sustav $LKID^{\omega}$ tako što svako (potencijalno beskonačno) dokazno stablo smije imati *najviše konačno mnogo* različitih podstabala te svaka beskonačna grana tog stabla mora imati beskonačno mnogo „napredaka". Dokazni sustav $CLKID^{\omega}$ prikazat ćemo neformalno.

Primjer 22. Dokazujemo tvrdnju $\forall x, Nat(x) \to Even(x) \vee Odd(x)$ u sustavu $CLKID^{\omega}$. Koristimo pokrate kao u primjeru 20.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{Nx \vdash Ex, Ox\,(\dagger)}{Ny \vdash Ey, Oy}\,(Subst)}{Ny \vdash Oy, Ey}\,(Perm)}{Ny \vdash Oy, Osy}\,(\texttt{PA\_prod\_O\_succ}Prod)}{Ny \vdash Esy, Osy}\,(\texttt{PA\_prod\_E\_succ}Prod)}{x = sy, Ny \vdash Ex, Ox}\,(EqL)}{\vdash Eo, Oo}\,(\texttt{PA\_prod\_E\_zero}Prod)}{\dfrac{\dfrac{\dfrac{Nx \vdash Ex, Ox\,(\dagger)}{Nx \vdash Ex \vee Ox}\,(OrR)}{\vdash Nx \to Ex \vee Ox}\,(ImpR)}{\vdash \forall x, Nx \to Ex \vee Ox}\,(AllR)}\,(Case\,N)$$

Iako zapisan u konačnom stablu, ovaj dokaz je beskonačan. Čitajući dokaz od dna do vrha, primjećujemo nove oznake te nova pravila. Za početak, vidimo da su znakom † označene gornja i donja sekventa $Nx \vdash Ex, Ox$. Gornju sekventu nazivamo *pupoljkom* (*bud*), a donju nazivamo *pratiteljem* (*companion*). Može se reći da su pupoljak i nje-

gov pratitelj povezani bridom, tvoreći ciklus u dokaznom stablu — odavde dolazi naziv *ciklički* dokaz. U stvari se isječak stabla dokaza između pratitelja i pupoljka ponavlja *ad infinitum* pa je dokaz beskonačne dubine. Nadalje, vidimo novo pravilo izvoda $Case$ koje odgovara rastavu po slučajevima, a koristi se umjesto pravila indukcije. Ako se između pupoljka i njegovog pratitelja javlja primjena pravila $Case$, kažemo da je napravljen *napredak*. Konačno, vidimo znak jednakosti uz kojeg je vezano lijevo pravilo za jednakost EqL. Znak jednakosti je za potrebe sustava $LKID^\omega$ i $CLKID^\omega$ (odnosno pravila $Case$) ugrađen u definiciju formule.[1]

Ciklički dokazi su oni dokazi beskonačne dubine u kojima je između svakog pupoljka i njegovog pratitelja napravljen napredak, odnosno primijenjeno je pravilo $Case$. U primjeru 22, pupoljak i njegov pratitelj javljaju se u istoj grani dokaza, no to nije nužno. Ovu pojavu možemo vidjeti na dokazu obrata sekvente $Nx \vdash Ex \vee Ox$.

Primjer 23. Dokazujemo sekventu $Ex \vee Ox \vdash Nx$ u sustavu $CLKID^\omega$.

$$\dfrac{\dfrac{\dfrac{}{\vdash No}\,(1)}{x = o \vdash Nx}\,(EqL) \quad \dfrac{\dfrac{\dfrac{Ox \vdash Nx\,(\dagger)}{Oy \vdash Ny}\,(Subst)}{Oy \vdash Nsy}\,(2)}{x = sy, Oy \vdash Nx}\,(EqL)}{Ex \vdash Nx\,(*)}\,(Case\,E) \quad \dfrac{\dfrac{\dfrac{Ex \vdash Nx\,(*)}{Ey \vdash Ny}\,(Subst)}{Ey \vdash Nsy}\,(3)}{x = sy, Ey \vdash Nx}\,(EqL)}{Ox \vdash Nx\,(\dagger)}\,(Case\,O)}{Ex \vee Ox \vdash Nx}\,(OrL)$$

Ovdje je znakom † označen par pupoljka i pratitelja sekvente $Ox \vdash Nx$, a znakom $*$ je označen par pupoljka i pratitelja sekvente $Ex \vdash Nx$. Iako se odgovarajući parovi pupoljaka i pratitelja naizgled javljaju u različitim granama, uz dva „odmatanja" stabla dokaza vidimo da to u stvari nije slučaj.

$$\dfrac{\dfrac{\dfrac{}{\vdash No}\,(1)}{x = o \vdash Nx}\,(EqL) \quad \dfrac{\dfrac{\dfrac{\dfrac{Ex \vdash Nx\,(*)}{Ey \vdash Ny}\,(Subst)}{Ey \vdash Nsy}\,(2)}{x = sy, Ey \vdash Nx}\,(EqL)}{Ox \vdash Nx\,\cancel{(*)}}\,(Case\,O)}{Ex \vdash Nx\,(*)}\,(Subst) \quad \dfrac{\dfrac{\dfrac{\dfrac{Ox \vdash Nx\,(\dagger)}{Oy \vdash Ny}\,(Subst)}{Oy \vdash Nsy}\,(2)}{x = sy, Oy \vdash Nx}\,(EqL)}{Ex \vdash Nx\,\cancel{(\dagger)}}\,(Case\,E)}{Ox \vdash Nx\,(\dagger)}\,(Subst)}{Ex \vee Ox \vdash Nx}\,(OrL)$$

Pupoljak sekvente $Ex \vdash Nx$ zamijenjen je isječkom stabla od pratitelja sekvente $Ex \vdash Nx$ do pupoljka sekvente $Ox \vdash Nx$, Nadalje, pupoljak sekvente $Ox \vdash Nx$ zamijenjen je

[1] Jednakost se u matematičkoj logici najčešće tretira kao predikatni simbol signature.

isječkom stabla od pratitelja sekvente $Ox \vdash Nx$ do pupoljka sekvente $Ex \vdash Nx$. Dokaz ne bi bio moguć bez korištenja produkcijskih pravila za produkcije:

(1) `PA_prod_N_zero` i

(2) `PA_prod_N_succ`.

Važno je spomenuti da je svojstvo „biti ispravan dokaz u sustavu $CLKID^{\omega}$" odlučivo te je na temelju toga napravljen ciklički dokazivač teorema CYCLIST[2] [19]. Za naš primjer proširene Peanove signature s produkcijama za induktivne predikatne simbole Nat, $Even$ i Odd, definicije produkcija možemo zapisati u konfiguracijsku datoteka `fo.defs` na idući način.

```
Nat {
    true    => Nat(0)  |
    Nat(x)  => Nat(s(x))
} ;
Even {
    true    => Even(0) |
    Odd(x)  => Even(s(x))
} ;
Odd {
    Even(x) => Odd(s(x))
} ;
```

Zatim možemo pokrenuti dokazivač naredbom

```
./fo_prove -D fo.defs -p -S "Nat(x) |- Even(x) \/ Odd(x)"
```

i dobiti izlaz programa

```
0: Nat_0(x) |- Odd_2(x) \/ Even_1(x) (Nat L.Unf.) [1 <{(0, 1)}/{}>, 2 <{(0, 1)}/{(0, 0)}>]
  1: Nat_1(0) |- Odd_2(0) \/ Even_1(0) (Even R.Unf.) [3 <{(1, 1)}/{}>]
    3: Nat_1(0) |- T \/ Odd_2(0) (Id)
  2: Nat_1(s(w)) /\ Nat_0(w) |- Odd_2(s(w)) \/ Even_1(s(w)) (Even R.Unf.) [4 <{(1, 1), (0, 0)}/{}>]
    4: Nat_1(s(w)) /\ Nat_0(w) |- Odd_4(w) \/ Odd_2(s(w)) (Odd R.Unf.) [5 <{(1, 1), (0, 0)}/{}>]
      5: Nat_1(s(w)) /\ Nat_0(w) |- Even_5(w) \/ Odd_4(w) (Weaken) [6 <{(0, 0)}/{}>]
        6: Nat_0(w) |- Odd_2(w) \/ Even_1(w) (Subst) [7 <{(0, 0)}/{}>]
          7: Nat_0(x) |- Odd_2(x) \/ Even_1(x) (Backl) [0] <pre={(0, 0)}>
```

[2] http://www.cyclist-prover.org/

koji predstavlja dokaz sekvente $Nat(x) \vdash Even(x) \vee Odd(x)$. Možemo primijetiti da je ovaj generirani dokaz sličan našem dokazu iz primjera 22.

6. Zaključak

Cilj ovog rada bio je ponuditi pregled visoke razine na alat za dokazivanje Coq te ga primijeniti na formalizaciju logike prvog reda s induktivnim definicijama. U tu svrhu smo u prvom poglavlju prikazali osnove Coqa i teorije na kojoj se on temelji, dok smo ostala poglavlja posvetili logici. Definirali smo osnovne pojmove poput formule, strukture, standardnog modela te dokaznog sustava i dokazali smo neka njihova svojstva. Kroz cijeli rad ilustrirali smo definicije i leme neformalnim i formaliziranim primjerima. Iako je ponekad bilo teže razlučiti bitne dijelove neformalne definicije, ipak se pokazalo da je Coq prikladan alat za formalizaciju logike te autor smatra da je cilj rada uspješno postignut.

Ovim radom smo tek zagrebali površinu formalizacije logike prvog reda s induktivnim definicijama. Prirodni nastavak rada je dokaz potpunosti sustava $LKID$ pomoću Henkinovih modela, a nakon toga i formalizacija cikličkog dokaznog sustava $CLKID^{\omega}$. Nadalje, ima smisla formalizirati dokazivače teorema za sustave $LKID$ i $CLKID^{\omega}$. Kako logika prvog reda nije odlučiva, takvi dokazivači teorema trebali bi uvesti dodatne pretpostavke (kao što je pretpostavka zatvorenog svijeta u Prologu) ili se usmjeriti prema interaktivnom dokazivanju. Za našu formalizaciju sustava $LKID$, ima smisla iskoristiti dodatne Coqove mogućnosti, posebno za automatizaciju dokaza jezikom taktika Ltac te za prikladniju notaciju terma, formula i dokaza.

Po autorovu mišljenju, računalna formalizacija matematike je odličan način razmišljanja i razumijevanja, posebno kada se koristi zajedno s tradicionalnim učenjem jer tada pojam koji se proučava postaje jasan u potpunosti — formalizacija nas tjera da budemo posebice pažljivi. Nadalje, u današnje vrijeme umjetne inteligencije i automatski generiranog koda, dokazivanje točnosti programa postaje bitnije no što je ikad bilo. Za kraj ćemo reći da je formalizacija proces koji je često naporan ali uvijek ispunjujuć.

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.

[19] J. Brotherston, N. Gorogiannis, i R. L. Petersen, “A Generic Cyclic Theorem Prover”, u *Programming Languages and Systems*, R. Jhala i A. Igarashi, Ur. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012., str. 350–367.

Sažetak

Primjene Coq alata za dokazivanje u matematici i računarstvu

Miho Hren

Coq je interaktivni dokazivač teorema temeljen na teoriji tipova. U ovom radu primijenili smo alat Coq na zadatak formalizacije logike prvog reda s induktivnim definicijama FOL_{ID}. Nadalje, definirali smo standardne modele i dokazni sustav $LKID$ za logiku FOL_{ID} te dokazali neka njihova svojstva. Također smo neformalno prikazali cikličke dokaze u kontekstu logike FOL_{ID}. Sve definicije i leme u radu ilustrirali smo primjerima te formalizacijom u Coqu.

Ključne riječi: Coq; logika prvog reda; induktivne definicije; dokaz

Abstract

Applications of the Coq Proof Assistant in mathematics and computer science

Miho Hren

Coq is an interactive theorem prover based on type theory. In this thesis we applied the Coq Proof Assistant to the task of formalizing first order logic with inductive definitions FOL_{ID}. Furthermore, we defined standard models and the $LKID$ proof system for FOL_{ID}, and we proved some of their properties. We also informally presented cyclic proofs in the context of FOL_{ID}. We illustrated all the definitions and lemmas in this thesis with examples and formalisations in Coq.

Keywords: Coq; first order logic; inductive definitions; proof