Uvod u interaktivno dokazivanje teorema

Sana Stojanović Đurđević, Filip Marić

April 12, 2021

Sadržaj

1

Uvod

theory *Cas0-vezbe*
  imports *HOL−Library.LaTeXsugar*

begin

Knjiga koja je pred vama nastala je kao nastavni materijal za kurs *Uvod u interaktivno dokazivanje teorema* na Matematičkom fakultetu (sa četvrte godine smera Informatika), koje smo držali tokom akademskih godina 2018/2019, 2019/2020 i 2020/2021. Iako je interaktivno dokazivanje moguće prikazati korišćenjem raznih dokazivača, tokom kursa se koristi isključivo interaktivni dokazivač teorema Isabelle/HOL. Jedan od razloga za ovaj izbor je to što Isabelle/HOL kroz svoju veoma bogatu notaciju i sintaksu i deklarativan jezik za zapis dokaza čini dokazivanje u njemu donekle blisko klasičnoj matematici, koja je bliska studentima. Dodatno, Isabelle/HOL nudi i bogatu podršku za funkcionalno programiranje i formalnu verifikaciju softvera.

Kompletna knjiga je kreirana u okviru samog interaktivnog dokazivača Isabelle. Sva poglavlja su Isabelle/HOL teorije, koje sadrže definicije, leme i njihove dokaze i mogu se pojedinačno učitati i pokrenuti tj. proveriti u dokazivaču Isabelle. Ovakav pristup kreiranju materijala bio je znatno zahtevniji za autore i posebna pažnja je posvećena usklađivanju kôda koji se prevodi u Isabelle-u sa tekstom kojim opisujemo njegovo značenje. Ovakav način kreiranja knjige nudi čitaocu jedinstvenu mogućnost da sam testira postojeći materijal (bez pojedinačnog prekucavanja i izdvajanja delova kôda koji ga zanima), da direktno iz njega uči, da ga dopunjuje, menja i savladava korak po korak od početnog poglavlja.

Prvo izdanje knjige je elektronski dostupno sa internet adrese: www.matf.bg.ac.rs/~sana/uidt.htm. Nadamo se da će time biti omogućem pristup širem skupu čitalaca i eventualno dovesti do veće popularnosti korišćenja

računara u svrhu dokazivanja teorema i formalne verifikacije softvera.

Elektronski materijali, koji se mogu koristiti da korisnik samostalno izgeneriše PDF verziju knjige na osnovu izvornog Isabelle/HOL koda, se nalaze na istoj adresi. Elektronski materijal je podeljen po poglavljima i dostupan je kako u Isabelle *.thy* formatu, tako i u *.htm* formatu. Iz tog razloga će mali deo ovog materijala biti posvećen i kreiranju izlaznih dokumenata na osnovu unapred kreirane teorije.

end

2

Interaktivni dokazivač teorema Isabelle

2.1 Dokazivanje teorema uz pomoć računara

theory *Cas1-vezbe*
imports *Main Complex-Main*

begin

Dokazivanje teorema uz pomoć računara inicijalno se razvijalo u pravcu *automatskog dokazivanja teorema*. Problem dokazivanja određene teoreme se sastojao u formulisanju tvrđenja u jeziku korišćenog programa za dokazivanje teorema i prepuštanje tom dokazivaču da potvrdi ili opovrgne da li je dato tvrđenje teorema unutar nekog fiksiranog formalnog sistema. Najčešće je izlaz dobijen od dokazivača nakon izvršavanja bio veoma ograničen, često je sadržao samo da/ne odgovor, tako da korisnik nije mogao da dobije mnogo informacija o samom dokazu tvrđenja. Automatsko dokazivanje teorema se veoma uspešno primenjuje u mnogim specijalizovanim matematičkim teorijama. Na primer, automatski dokazivači specijalizovani za iskaznu logiku se uspešno koriste u verifikaciji hardvera, jer su u stanju da rezonuju o formulama koje imaju stotine hiljada promenljivih i milione ograničenja. Automatski dokazivači teorema za geometriju su zasnovani na svođenju na analitičku geometriju i na algebrizaciji geometrijskih tvrđenja i u stanju su da uspešno dokažu pregršt komplikovanih geometrijskih tvrđenja.

Ipak, automatsko dokazivanje teorema je veoma izazovno i čak ni najsavremeniji automatski dokazivači nisu u stanju da potpuno samostalno, bez ljudskog navođenja dokazuju kompleksne teoreme iskazane u bogatim matematičkim teorijama. Stoga se poslednjih nekoliko decenija uz automatsko dokazivanje, sve više se koristi i *interaktivno dokazivanje teorema*, koje podrazumeva saradnju između čoveka i računara u procesu dokazivanja. Čovek zapisuje dokaz u specijalizovanom jeziku, prepuštajući računaru da samostalno pronađe jednostavnije korake u dokazu. Dakle, za razliku od procesa automatskog dokazivanja teorema, gde je korisnik nakon formulisanja teoreme samo posmatrač, dok dokazivač sam dokazuje teoremu i generiše izlaz u određenom formatu, proces interaktivnog dokazivanja teorema u mnogo većoj meri zavisi od korisnika. Korisnik sam formuliše i unosi glavne korake dokaza, a računar proverava da li je uneti dokaz formalno ispravan, dopunjujući automatski jednostavnije, nedostajuće delove.

Neki od najčešće korišćenih interaktivnih dokazivača teorema danas su:

- Isabelle[1],

[1] http://www.cl.cam.ac.uk/research/hvg/Isabelle/

- Coq[2],
- Lean[3],
- Mizar[4], i
- HOL-Light[5].

Danas je česta saradnja između tradicionalnih automatskih i interaktivnih dokazivača i interaktivni dokazivači imaju mogućnost korišćenja nekih automatskih dokazivača. Naime, iako automatski dokazivači najčešće ne generišu kompletan formalni dokaz, oni mogu da daju informacije o skupu aksioma i pomoćnih lema koje su korišćene tokom dokazivanja, koje interaktivni dokazivač može iskoristiti da rekonstruiše i proveri celokupan formalni dokaz.

Interaktivni dokazivači teorema se smatraju mnogo pouzdanijim od tradicionalnih automatskih dokazivača, jer insistiraju na tome da se svaki dokaz izvede potpuno detaljno u zadatom formalnom sistemu. U krajnoj instanci (koja može biti sakrivena od korisnika) svaki dokaz se izražava u obliku primena osnovnih aksioma i pravila izvođenja i tzv. pouzdano jezgro dokazivača (engl. trusted core) proverava korektnost svakog dokaza. Dokaz se prihvata isključivo ako ga pouzdano jezgro uspešno proveri. Obratimo pažnju na to da sve napredne komponente interaktivnog dokazivača koje pronalaze dokaze ne moraju biti verifikovane, jer ako one sadrže neku grešku i pronađu dokaz koji nije ispravan, pouzdano jezgro će to uočiti i takav dokaz će biti odbačen.

Otkrivanje grešaka u matematičkim dokazima, u udžbenicima i u objavljenim radovima, i u programima koji se koriste u životno ključnim situacijama, dovelo je do potrebe za mašinski proverivim dokazima teorema i formalnom verifikacijom napisanih programa. Osim primene u obrazovanju i u industriji, interaktivni dokazivači teorema značajno doprinose i očuvanju bogatog matematičkog nasleđa kroz formalizaciju bitnih matematičkih dela.

Dva najčešće korišćena pristupa koja se koriste prilikom interaktivnog dokazivanja teorema su proceduralni i deklarativni pristup[6].

[2] http://coq.inria.fr/
[3] https://leanprover.github.io/
[4] http://www.mizar.org
[5] http://www.cl.cam.ac.uk/~jrh13/hol-light/
[6] Freek Wiedijk. A Synthesis of the Procedural and Declarative Styles of Interactive Theorem Proving. Logical Methods in Computer Science, 8(1), 2012.

Proceduralni pristup intenzivno koristi automatske alate ili taktike sa zadatkom pojednostavljivanja trenutnog cilja teoreme. Pažljivim kombinovanjem taktika cilj teoreme se pojednostavljuje dok se ne dođe do pretpostavki teoreme, kada možemo reći da je dokaz završen. Dakle, u ovom pristupu, dokaz se sastoji od niza poziva taktika koje transformišu teoremu koja se dokazuje i svode je na jednostavnija tvrđenja.

Prilikom korišćenja *deklarativnog pristupa*, dokaz se zapisuje na jeziku koji podseća na standardni tekst koji se javlja u matematičkim udžbenicima, zahvaljujući čemu su dobijeni dokazi nalik dokazima iz matematičkih udžbenika i po strukturi i po sintaksi. Deklarativni dokazi su zbog toga pristupačniji prosečnom matematičaru i studentu, za razliku od proceduralnih dokaza koji zahtevaju malo veće poznavanje jezika nad kojim su napisani.

2.2 Istorijat Isabelle-a

Interaktivni dokazivač teorema Isabelle se razvija od sredine 1980-tih godina. Kao najznačajniji autori, sa najviše doprinosa, ističu se Lari Polson (eng. *Larry Paulson*) sa Univerziteta u Kembridžu i Tobijas Nipkov (eng. *Tobias Nipkow*) i Markus Vencel (nem. *Markus Wenzel*) sa Tehničkog univerziteta u Minhenu. Isabelle je u određenoj meri zasnovan na dokazivaču LCF i može se reći da prati LCF pristup (eng. *LCF approach*) [**?**] kako bi se obezbedila korektnost sistema (on podrazumeva postojanje proverenog jezgra i specifičnu upotrebu funkcionalnog programskog jezika kojom se obezbeđuje pouzdana provera dokaza bez potrebe da ceo dokaz bude eksplicitno konstruisan i smešten u memoriju računara).

Isabelle je generički dokazivač i omogućava rad sa različitim formalnim sistemima (tzv. objektnim logikama). U okviru Isabelle-a implementirana je *meta-logika* koja se naziva Isabelle/Pure u okviru koje se definišu različite objektne logike. Isabelle podržava više objektnih logika među kojima su najznačajnije:

- *HOL* logika višeg reda,
- *FOL* klasična i intuicionistička logika prvog reda,
- *ZF* Cermelo-Frenkelova (eng. *Zermelo-Fraenkel*) teorija skupova,
- *CTT* konstruktivna teorija tipova,
- *LCF* Skotova logika izračunljivih funkcija u logici prvog reda,

- *HOLCF* Skotova logika izračunljivih funkcija u logici višeg reda.

Objektna logika sistema Isabelle koja je ubedljivo najkorišćenija je logika višeg reda (eng. *Higher Order Logic*). Verzija dokazivača Isabelle koja koristi logiku višeg reda se naziva *Isabelle/HOL* [**?**] i ona se pojavila početkom 1990-tih. Ova verzija će biti zastupljena u ovoj knjizi. Logika višeg reda u dokazivaču Isabelle/HOL zasnovana je na tipiziranoj verziji lambda-računa (engl. typed lambda-calculus), ali se u ovoj knjizi nećemo baviti logičkim osnovama sistema, već samo njegovom praktičnom primenom.

Osim za dokazivanja matematičkih teorema i formaliziciju matematičkih teorija, Isabelle/HOL se intenzivno koristi i kao alat za verifikaciju programa. Naime, Isabelle/HOL nudi funkcionalni programski jezik u kom je moguće definisati funkcije koje implementiraju različite algoritme, a zatim formulisati i dokazati njihova svojstva. Na osnovu tih definicija, moguće je automatski izvesti programski kod u više funkcionalnih programskih jezika (Haskell, SML, OCaml) [**?**].

2.3 Instalacija i pokretanje Isabelle/HOL

Program za interaktivno dokazivanje teorerma Isabelle/HOL se može preuzeti sa stranice https://isabelle.in.tum.de/. Ova knjiga je napisana u verziji *Isabelle2021/HOL*. Na većini računara ovaj program se instalira jednostavno, ali ako postoji problem mogu se pratiti uputstva sa stranice https://isabelle.in.tum.de/installation.html.

Nakon pokretanja Isabelle/HOL otvara 3 osnovna okvira koje ćemo zvati: **teorija** (gornja polovina), **pregled** (skroz desno), i **izlaz** (donja polovina). Ako okvir *izlaz* nije otvoren treba pritisnuti dugme `Output` na dnu ekrana. Ako okvir *pregled* nije otvoren treba pritisnuti dugme `Sidekick` sa desne strane. Dodatno može se pristupiti bogatoj Isabelle/HOL dokumentaciji tako što se sa leve strane programa pritisne dugme `Documentation` Ovoj dokumentaciji se može pristupiti i online sa adrese http://isabelle.in.tum.de/documentation.html, kao i iz foldera u kom je Isabelle/HOL instaliran (u podfolderu *doc*).

U okvir za *teoriju* unose se sve teorije, definicije i leme koje se dokazuju. Okvir *izlaz* prikazuje sve tekuće ciljeve programa i u svakom koraku prikazuje trenutno stanje programa. Okvir *pregled* prikazuje trenutnu strukturu dokumenta koji pišemo u okviru *teorija*.

2.4 Primer teorije u Isabelle/HOL

Pre nego što započnemo sistematičan pregled sistema Isabelle/HOL, uvedimo na jednom jednostavnom primeru njegove osnovne pojmove. Osnovna jedinica sadržaja u Isabelle/HOL je teorija. U teoriji se uvode definicije i formulišu se i dokazuju tvrđenja. Iako u matematici razlikujemo leme, teoreme, stavove i slično, u formalnom sistemu nema nikakve razlike između njih (sve to su tvrđenja koja se na isti način formulišu, dokazuju i kasnije koriste). Razmotrimo jedan veoma jednostavan primer teorije.

theory *Sledbenik*
imports *Main*
begin

definition *sledbenik* :: *nat* ⇒ *nat* where
sledbenik x = *x* + *1*

lemma *sledbenik* (*sledbenik x*) = *x* + *2*
unfolding *sledbenik-def*
by auto

lemma
assumes *x* > *0* ∧ *x* < *3*
shows *sledbenik x* > *1* ∧ *sledbenik x* < *4*
using *assms*
unfolding *sledbenik-def*
by auto

end

Kretanjem kursorom kroz teoriju, možemo videti kako Isabelle/HOL tumači liniju na kojoj se kursor trenutno nalazi.

Napomena: Da bi mogla da se prati interpretacija svake naredbe u Isabelle-u potrebno je sa desne strane, u delu *pregled*, selektovati dugme *Theories* i selektovati opciju *Continuous checking*. U okviru tog prozora se prikazuju sve trenutno otvorene teorije.

Napomena: Dodatno, da bi se video izlaz u Isabelle/HOL sistemu potrebno je otvoriti *Output* prozor na dnu ekrana i u njemu štiklirati opciju *Proof state*.

Na početku je definisana funkcija *sledbenik*. Isabelle/HOL je tipizirani jezik i svi objekti imaju svoj tip. Tip ove funkcije je $nat \Rightarrow nat$, što znači da je ovo funkcija koja slika prirodne brojeve u prirodne brojeve. Nakon navođenja tipa data je jednakost kojom se definiše funkcija *sledbenik*. Primetimo da se, u skladu sa praksom funkcionalnog programiranja, poziv tj. primena funkcije piše bez zagrada (umesto *sledbenik*(x), piše se *sledbenik* x). Više reči o sintaksi biće dato u nastavku ovog poglavlja.

U nastavku su formulisane i dokazane dve leme. Formulacija prve leme je očigledna. Dokazuje se u potpunosti primenom automatskog dokazivača (koji je pozvan pomoću *by auto*). Naredbom *unfolding sledbenik-def* postignuto je da se pre primene automatskog dokazivača primeni definicija funkcije *sledbenik*. Ovo je potrebno posebno navesti, jer automatski dokazivači po pravilu ne koriste definicije (u suprotnom bi se od teoreme koja se dokazuje veoma brzo došlo do najjosnovnijih pojmova na kojima je zasnovan ceo formalni sistem i teoreme bi postale previše glomazne i komplikovane).

I naredna lema je veoma očigledna. Konstrukcijom *assumes* − *shows* formulisane su pretpostavka i zaključak leme. Prva lema je imala samo zaključak, pa je bilo moguće izostaviti ključnu reč *shows*, mada ne bi bila greška ni da je bila navedena. Unutar pretpostavke i unutar zaključka upotrebljen je veznik konjunkcije (o tome kako se pomoću tastature unose matematički simboli, biće reči u nastavku). Pomoću *using assms*, automatskom dokazivaču je naglašeno da prilikom dokazivanja leme koristi i pretpostavku (podrazumevano ponašanje automatskih dokazivača je to da ne pokušavaju da koriste pretpostavke sve dok ih korisnik eksplicitno ne navede).

2.5 Sintaksa jezika Isabelle/HOL

U Isabelle/HOL sistemu razlikujemo *unutrašnju* i *spoljašnju* sintaksu. Fiksirana sintaksa koja se koristi prilikom definisanja teorije (čiji su deo ključne reči **begin** i **datatype**) naziva se *spoljašnja sintaksa* (eng. *outer syntax*). *Unutrašnja sintaksa* (eng. *inner syntax*) je definisana samim jezikom teorije koju korisnik želi da kreira (opis teorije, njenih definicija, lema i njihovih dokaza).

Spoljašnja sintaksa definiše sintaksu koja se koristi prilikom pisanja svih komandi. Čine je ključne reči poput *lemma*, *datatype*, *assumes*, *proof*, *qed*, *show*, *have*, itd. i opisuje konstrukte koji su dozvoljeni u okviru jedne teorije. Njen detaljan opis se može naći u tutorijalu [**?**]. Dodatno nju

čine i imena tipova (*float*, *int*, *nat*), specijalni matematički simboli kao što su $\forall$ i λ (koji se u Isabelle/HOL zapisuju na uobičajen način `\<forall>` i `\<lambda>`), naredbe za zapisivanje teksta (koji se zapisuje u okviru naredbe `text\<open>\<close>`), nevidljivi komentari (koji se zapisuju u okviru naredbe `(* *)`) i vidljivi komentari (koji se zapisuju u okviru naredbe `\<comment> \<open>\<close>`). Ovi konstrukti i još mnogo njih će biti detaljno opisani kroz ovu knjigu.

Unutrašnja sintaksa se odnosi na tekst koji se nalazi između dvostrukih navodnika. Na primer, prilikom definisanja konkretne notacije koja se koristi u željenom logičkom okviru koji korisnik razvija (nakon ključnih reči kao što su npr. **definition**, **primrec**, **datatype**).

Prilikom zapisivanja *unutrašnje sintakse* korišćenje navodnika je neophodno, osim u slučaju pojedinačnih promenljivih (na primer, `"x+y"` i `z` su dozvoljeni, ali `x+y` zapisano bez navodnika nije)[7].

Prilikom prijavljivanja sintaksne greške Isabelle će ispisati *Inner syntax error* za grešku koja se javlja u okviru unutrašnje sintakse, odnosno *Outer syntax error* za grešku koja se javlja u okviru spoljašnje sintakse.

2.5.1 Osnovni pojmovi

U ovom poglavlju će biti prikazani osnovni matematički pojmovi i sintaksa korišćena za njihovo zapisivanje u jeziku Isabelle/HOL. Jezik koji se koristi kreiran je u duhu duge tradicije funkcionalnog programiranja.

U uvodnom primeru smo videli da je Isabelle/HOL tipiziran jezik (koristili smo tip prirodnih brojeva *nat*).

Osnovni tipovi podataka u jeziku Isabelle/HOL su: *bool*, *nat*, *int*, *real*, *complex* odgovaraju redom: tipu logičkih vrednosti, tipu prirodnih, celih realnih i kompleksnih brojeva.

Izvedeni tipovi podataka se kreiraju uz pomoć tzv. konstruktora tipova: na primer, *list* za listi i *set* za kreiranje skupova elemenata nekog tipa. Ova dva konstruktora se pišu postfiksno, npr. lista prirodnih brojeva se zapisuje *nat list*, a skup realnih brojeva *real set*.

Tip funkcija zadaje se operatorom $\Rightarrow$. Na primer, tip $nat \Rightarrow real$ označava niz realnih brojeva (funkciju koja svakom indeksu koji je prirodan broj pridružuje neku realnu vrednost). U situaciji kada funkcija ima više ulaznih parametara, koristi se Karijev oblik funkcije (eng. *currying*) i zapisuje se

[7] Treba imati na umu da prilikom generisanja pdf dokumenata ovi navodnici neće biti vidljivi.

u obliku $nat \Rightarrow nat \Rightarrow real$. Ovaj oblik funkcije je fleksibilniji i ima dosta prednosti naspram zapisa $nat \times nat \Rightarrow real$ koji koristi torke, od kojih je možda najbitnija ta da se dobro slaže sa principom matematičke indukcije. U Isabelle/HOL (slično kao u Haskell-u) osim kada torke nisu eksplicitno zahtevane funkcije se uvek definišu kao Karijeve funkcije.

Tipske promenljive - se koriste za zapis polimorfnih funkcija, odnosno kada imamo potrebu da umesto konkretnog tipa navedemo promenljivu proizvoljnog tipa. Tipske promenljive navode se sa apostrofom ispred: $'a$, $'b$. Mnoge standardne funkcije su definisane kao polimorfne. Na primer, funkcija koja izračunava dužinu liste ima tip $'a\ list \Rightarrow nat$, što znači da joj je argument lista elemenata bilo kog tipa, a da joj je rezultat prirodan broj.

Termi se kreiraju kao u funkcionalnim programskim jezicima. Funkcija f tipa $\tau_1 \Rightarrow \tau_2$ primenjena na term t tipa τ_1 će kreirati term $f\ t$ tipa τ_2. Ako želimo da naglasimo da će term t biti tipa τ to zapisujemo $t :: \tau$. Isabelle/HOL podržava i λ-funkcije, čije korišćenje će biti ilustrovano u kasnijim poglavljima. Dozvoljeno je korišćenje konstanti (funkcije arnosti 0) i promenljivih kao baznih terma.

Isabelle/HOL podržava i neke od osnovnih konstrukata preuzetih iz funkcionalnih programskih jezika (ova tri konstrukta se uvek navode između običnih zagrada).*let*-izraz uvodi vrednosti (t_1, t_2, ...) promenljivima (x_1, x_2, ...) koje se koriste u izrazu *u*. *case*-izraz definiše vrednost u zavisnosti od oblika izraza *t*. *if*-izraz je uslovni izraz koji predstavlja skraćenicu za *case*-izraz.

$$(let\ x_1 = t_1\ x_2 = t_2\ in\ u)$$
$$(case\ t\ of\ pat_1 \Rightarrow t_1 \mid \ldots \mid pat_n \Rightarrow t_n)\ (if\ b\ then\ t_1\ else\ t_2)$$

2.5.2 Zapisivanje formula u Isabelle/HOL

U ovom poglavlju biće prikazana osnovna Isabelle/HOL sintaksa koja se koristi za zapisivanje formula. U prvom trenutku se nećemo baviti samim dokazima zadatih tvrđenja već ćemo se osloniti na automatske dokazivače.

- *Formule* su izrazi nad tipom *bool* sa standardnim konstantama *True* i *False* i sa uobičajenim logičkim veznicima (navedenim po rastućem prioritetu): $\neg$, $\wedge$, $\vee$, $\longrightarrow$. Veznik $\longrightarrow$ ima desnu asocijativnost, što znači da naredni izraz $A \longrightarrow B \longrightarrow C$ ima značenje kao da su zagrade postavljene ovako $A \longrightarrow (B \longrightarrow C)$, dok ostali binarni veznici imaju levu asocijativnost.

- *Jednakost* se koristi uobičajeno kao infiksni operator = čiji je tip $'a \Rightarrow 'a \Rightarrow bool$, što znači da se može primeniti na objekte raznih tipova (ali međusobno jednakih), pa i nad formulama. Jednakost formula ima značenje "ako i samo ako", a može se zapisati i sa sa $\longleftrightarrow$ (operatori = i $\longleftrightarrow$ imaju isto logičko značenje kada se primene na logiče formule tj. terme tipa *bool*, ali operator $\longleftrightarrow$ ima mnogo manji prioritet).

- *Univerzalni i egzistencijalni kvantifikator* se zapisuju na uobičajen način $\forall\, x.\ P$ i $\exists\, x.\ P$.

  Formule tog oblika, u kojima se veznici primenjuju samo na promenljive tipa *bool* i u kojima se ne koriste kvantifikatori, nazivaćemo *iskazne formule*. Formule u kojima se kvantifikacija vrši samo nad promenljivima elementarnih tipova (npr. nad brojevima) nazivaćemo *formule logike prvog reda*, a formule u kojima se kvantifikacija vrši nad funkcijama nazivaćemo *formule logike višeg reda*. Na primer, formula $\neg\ (A \wedge B) \longleftrightarrow \neg\ A \vee \neg\ B$ je iskazna formula, formula $\neg(\forall\ x.\ P\ x) \longleftrightarrow (\exists\ x.\ \neg\ P\ x)$ je formula logike prvog reda, dok je formula $(\forall\ x.\ \exists\ y.\ P\ x\ y) \longleftrightarrow (\exists\ f.\ \forall\ x.\ P\ x\ (f\ x))$ formula logike višeg reda.

  Najčešće korišćeni logički simboli i njihov ASCII zapis (u proširenom i u skraćenom obliku) su navedeni u nastavku teksta. Ovi (i slični) simboli se mogu ispisati u obliku naredbe ili se učitati prilikom kucanja naredbe (pritiskom na dugme *Tab* kada sistem prepozna i ponudi nekoliko mogućih simbola) ili, u situacijama kada se učitava ranije napisan Isabelle/HOL fajl, biranjem opcije iz menija *File/Reload with Encoding/UTF−8−Isabelle* što će ponovo učitati simbole umesto ASCII zapisa simbola.

$\forall$	\<forall>	ALL
$\exists$	\<exists>	EX
λ	\<lambda>	%
$\longrightarrow$	-->	
$\longleftrightarrow$	<->	
$\wedge$	/\	&
$\vee$	\/	|
$\neg$	\<not>	~
$\neq$	\<noteq>	~=

Za sve nabrojane veznike, u Isabelle/HOL možemo koristiti naredbu *term* da dobijemo tip veznika. Kada se ispituje tip, veznik mora da bude naveden u zagradi, npr. *term* $(\wedge)$: $bool \Rightarrow bool \Rightarrow bool$.

term ($\wedge$)

Prilikom formulisanja teorija, definicije i tvrđenja zapisuju se korišćenjem do sada opisane sintakse. Ta sintaksa predstavlja sintaksu objektnog nivoa, a za formule koje smo do sada videli kažemo da su formule objektnog nivoa u logici višeg reda Isabelle/HOL. Interno, međutim, ova objektna logika je samo jedna od raznih logika koje sistem Isabelle poznaje. Sve te logike (uključujući i ovu logiku Isabelle/HOL) definisane su u okviru meta-logike sistema Isabelle.

2.5.3 Meta-logika

Meta-logika predstavlja fragment intuicionističke logike višeg reda. Ponekad se naziva i Isabelle/Pure [**?**]. Meta-logika u svojoj sintaksi podržava dva osnovna simbola: meta-univerzalni kvantifikator $\bigwedge$ i meta-implikaciju $\Longrightarrow$, i jednakost $\equiv$. Nestandardne oznake se koriste da bi objektne logike imale na raspolaganju standardne oznake i veznike. Primetimo da je meta-logika deo sistema Isabelle, a ne deo HOL logike (u HOL logici koristimo standardne simbole $\forall$ i $\longrightarrow$, o čemu će biti više reči kasnije).

Neformalno govoreći, kvantifikator $\bigwedge m.$ "označava proizvoljnu ali fiksiranu promeljivu m" i biće intenzivno korišćen u dokazima u nastavku knjige.

Meta-implikacija, odnosno veznik $\Longrightarrow$ se koristi da odvoji pretpostavke od zaključka tvrđenja koje se dokazuje (i ima ulogu simbola $\vDash$ u računu sekvenata). On takođe (kao i $\longrightarrow$) ima desnu asocijativnost tako da formula $A1 \Longrightarrow A2 \Longrightarrow A3$ ima značenje kao da je napisana $A1 \Longrightarrow (A2 \Longrightarrow A3)$. Pošto se ovaj zapis tvrđenja koristi kada sistem Isabelle prikazuje korisniku tvrđenja koja su dokazana (u ovom slučaju $A1$ i $A2$) i tvrđenja koja treba dokazati (u ovom slučaju $A3$), moguće je koristiti malo čitljiviju Isabelle notaciju. Zapis $[\![A1; ...; An]\!] \Longrightarrow A$ predstavlja skraćen zapis za niz implikacija $A1 \Longrightarrow ... \Longrightarrow An \Longrightarrow A$ [8].

Tako, na primer, zapis $[\![A;\ A \longrightarrow B]\!] \Longrightarrow B$ označava da se pod pretpostavkama A i $A \longrightarrow B$ dokazuje zaključak B (ovo je, prepoznaćete, klasičan modus ponens u iskaznoj logici).

Pored toga, korisnici mogu koristiti meta-logiku u zapisu svojih tvrđenja na sledeći način:

lemma $[\![A;\ A \longrightarrow B]\!] \Longrightarrow B$

[8] Da bi se dobio ovakav skraćeni ispis u sistemu Isabelle potrebno je podesiti opciju *Plugins/Plugin Options/Isabelle/General > Print Mode* na vrednost *brackets* i restartovati sistem.

ali češće se ovakva tvrđenja zapisuju korišćenjem kombinacije *assumes*/*shows* (koja je deo jezika Isar koji će naknadno biti objašnjen).

lemma
assumes A and $A \longrightarrow B$
shows B

2.5.4 Kreiranje teorije

Kao što smo u primeru videli, datoteka koji sadrži definicije tipova, konstanti, funkcija i teoreme se naziva *teorija*. Teorija sa imenom T mora biti sačuvana u datoteci sa imenom $T.thy$ (puno ime teorije koje ne sme sadržati razmake, nakon čega sledi ekstenzija *.thy*).

Na početku dokumenta navodi se ključna reč *theory* nakon koje navodimo ime samog dokumenta bez ekstenzije. Ključnom rečju *imports* možemo uključiti dodatne teorije koje su nam neophodne za prevođenje tekuće teorije. Početak i kraj same teorije su označeni ključnim rečima *begin* i *end*, između kojih se navode definicije, teoreme i njihovi dokazi.

theory T
imports $T_1 \ldots T_n$
begin
definitions, theorems and proofs
end

U ovom primeru sa $T_1 \ldots T_n$ su označene postojeće teorije koje se koriste za građenje tekuće teorije T. Nazivamo ih roditeljskim teorijama i sve definicije i teoreme iz tih teorija su vidljive i u tekućoj teoriji. U većini primera biće uključena teorija *Main* koja u sebi sadrži uniju osnovnih teorema nad listama, skupovima i aritmetičkim operacijama.

Komentari se zapisuju između (* i *) ili u okviru naredbe `\<comment>\<open>\<close>` koja se prikazuje kao — . Prva vrsta komentara nije vidljiva u pdf dokumentu, tako da će komentari u ovoj knjizi biti prikazani na drugi način (pogledajte primer za uvođenje sinonima).

Teoreme u okviru sistema Isabelle/HOL mogu biti označene ključnom rečju `theorem` ili `lemma`, na korisniku je da izabere koju oznaku će koristiti (interno nema ama baš nikakve razlike). Ako u samoj teoriji imamo više pomoćnih tvrđenja i jedno glavno tvrđenje, obično će samo to glavno tvrđenje biti označeno ključnom rečju `theorem`. Teorema se zapisuje logičkim veznicima, npr. $A \longrightarrow A \vee B$.

Definicijama se definišu konstante i (nerekurzivne) funkcije. Svaka definicija se zadaje u obliku definicione jednakosti. Na primer, funkciju koja računa kvadrat broja možemo uvesti definicijom na sledeći način.

definition *definicija-kvadrat* :: *nat* ⇒ *nat* **where**
definicija-kvadrat n = *n* ∗ *n*

Kao što smo u uvodnom primeru videli, definicije se u dokazima ne raspisuju (*engl. unfold*) automatski i, ako je u dokazu neke leme potrebno koristiti definicionu jednakost, definicija mora biti eksplicitno raspisana. Kao što je već rečeno, korišćenjem ključne reči *unfolding* nakon koje se navodi ime definicije na koje se dopisuje `_def`. Ova naredba se primenjuje direktno na cilj koji se dokazuje i transformiše ga prema definiciji koja je navedena. Ako se definišu objekti tipa *bool*, tada se umesto obične jednakosti koristi simbol ⟷, koji ima niži prioritet, pa nije neophodno koristiti neke zagrade.

U sledećem primeru definišemo funkciju paran i, zbog nižeg prioriteta simbola ekvivalencije u odnosu na simbol jednakosti, ne moramo koristiti zagrade sa desne strane simbola ekvivalencija.

definition *paran* :: *nat* ⇒ *bool* **where**
paran n ⟷ *n mod 2* = *0*

Zbog povećanja čitljivosti i jednostavnijeg zapisivanja, Isabelle/HOL podržava i korišćenje sinonima i skraćenica.

Sinonime koristimo u slučaju da želimo da uvedemo novo ime za tip koji možemo da izrazimo uz pomoć konstruktora tipova. U kôdu se sinonimi automatski raspisuju i ne koriste se u internoj reprezentaciji tvrđenja u kome se javljaju. Služe isključivo udobnosti korisnika. Na primer tip *string* možemo uvesti na uobičajen način uz pomoć sinonima:

type-synonym *string* = *char list*

Skraćenice se uvode na sličan način kao definicije. Definicije se uvode simbolom jednakosti =, a skraćenice simbolom meta-ekvivalencije ≡. ASCII reprezenacija simbola ≡ je `==` ili \<equiv>.

abbreviation *skracenica-kvadrat′* :: *nat* ⇒ *nat* **where**
skracenica-kvadrat′ n ≡ *n* ∗ *n*

Za razliku od definicija, skraćenice se u dokazima automatski zamenjuju odgovarajućim oblikom čiji zapis skraćuju (nije potrebno raspisivati ih). Takođe, sistem prilikom prikaza formula koristi skraćenice gde god je to moguće. Ovo se može videti ako se pogledaju naredne dve leme, koje imaju isto značenje sa tom razlikom što se u prvoj koristi skraćenica (koja se automatski raspisuje), a u drugoj se koristi definicija (koju moramo eksplicitno raspisati u samom dokazu).

lemma *skracenica-kvadrat′ (n+1) = skracenica-kvadrat′ n + 2∗n + 1*
by *auto*

lemma *definicija-kvadrat (n+1) = definicija-kvadrat n + 2∗n + 1*
unfolding *definicija-kvadrat-def*
by *auto*

Napomena: Prilikom generisanja pdf dokumenta u sistemu Isabelle/HOL ne prikazuju se dvostruki navodnici koji se koriste prilikom zapisivanje formula koje pripadaju *unutrašnjoj sintaksi*. Najčešće je taj zapis zamenjen *italic tekstom*. Na primer, u prethodnom primeru, tip definicije mora biti pod navodnicima, kao i sama formula kojom se ta definicija uvodi. Prilikom navođenja lema, tvrđenje koje se dokazuje mora biti navedeno pod dvostrukim navodnicima.

Još jedno odstupanje je u prikazivanju simbola podvlake (_) koji se prikazuje kao crtica, ovako -, pa se na primer naziv definicije koji u sistemu Isabelle/HOL izgleda okako: `definicija_kvadrat`, u knjizi prikazuje ovako: *definicija-kvadrat*. Ovo važi za sva pojavljivanja podvlake.

2.5.5 Automatski alati

Iako je program Isabelle/HOL prvenstveno namenjen interaktivnom dokazivanju teorema, alati za automatsko dokazivanje su neophodni i postoji nekoliko metoda različite snage. Najčešće korišćeni metodi su metod za automatsko dokazivanje *auto* i simplifikator *simp*.

Metod *auto* pokušava da u jednom koraku dokaže tekuće tvrđenje. U većini realnih situacija ovo neće biti moguće tako da se najčešće koristi za dokazivanje jednostavnih logičkih tvrđenja. U slučaju da je tvrđenje teoreme podeljeno na više ciljeva, svi ciljevi se dokazuju odjednom, pretežno kroz njihovo pojednostavljivanje ali na raspolaganju ima i određen skup već dokazanih lema koje se nalaze u pomoćnim teorijama koje su uključene na početku fajla.

Metod *simp* je glavni deo metoda *auto* ali je dosta slabiji od njega i može se reći da mu je ponašanje predvidljivije. Često se koristi za dokazivanje jednostavnih linearnih tvrđenja. U nekim situacijama dešava se da metod *auto* zbog svoje moći ne uspeva da se snađe sa tekućim ciljem i da metod *simp* uspeva u slučaju kada on ne može. Može se koristiti ili nezavisno, naredbom `by simp` ili `apply simp`, ili u kombinaciji sa dodatnim lemama, naredbom `apply (simp add: th_1... th_n)`.

Ovaj mehanizam nam pruža jednostavan način za dodavanje definicija u

simplifikator, pošto je podrazumevano ponašanje u sistemu Isabelle/HOL da se definicije ne dodaju automatskim alatima. Na primer definicija sa imenom *f* bi se dodala u simplifikator naredbom `apply (simp add: f_def)`. Kada se metod *auto* koristi sa dodatnim lemama one se navode nakon ključnih reči *simp add*: `apply (auto simp add: th_1... th_n)`.

Nekoliko jednostavnih tvrđenja koja se mogu dokazati i sa *auto* i sa *simp*:

lemma $A \longleftrightarrow A \wedge A \wedge A$
by *simp*

lemma $A \longrightarrow A \vee B$
by *auto*

Trivijalno tvrđenje koje koristi malopre definisane skraćenice i definicije se ne dokazuje isto. Tvrđenje formulisano sa skraćenicom se automatski dokazuje bez problema, ali tvrđenje formulisano sa definicijom neće proći automatski već se mora eksplicitno navesti definicija kojom je predikat koji se koristi uveden.

lemma *skracenica-kvadrat'* $x = x * x$
by *auto* — Prolazi!

```
(* Naredni dokaz ne prolazi! *)
lemma "definicija_kvadrat x = x * x"
  by auto
```

— Dokaz prolazi uz pomoć naredbe *auto simp add*.
lemma *definicija-kvadrat* $x = x * x$
by (*auto simp add*: *definicija-kvadrat-def*)

— Dokaz prolazi uz pomoć naredbe *unfolding*.
lemma *definicija-kvadrat* $x = x * x$
unfolding *definicija-kvadrat-def*
by *auto*

Osnovne karakteristike metoda *simp* i *auto* su:

- Njihova primena pokazuje gde se sistem zaglavio, što može biti korisno za otkrivanje pomoćnih lema koje mogu pomoći u dokazivanju originalnog tvrđenja (što će biti prikazano u narednim poglavljima).
- Koriste se za primenu jednostavnih koraka ali su veoma nekompletni metodi.

Za metod dokazivanja kažemo da je *kompletan* ako je u stanju da dokaže svaku tačnu formulu. Međutim, ne postoji kompletan metod dokazivanja za logiku višeg reda. Postoji nekoliko metoda koji se mogu koristiti pored metoda *auto*:

- Metod *fastforce* se primenjuje samo na prvi podcilj, ne transformiše tekući cilj dodatno već samo uspe da ga dokaže ili ne. Može se pokrenuti sa dodatnim lemama kao *auto*, korišćenjem naredbe *simp add*.
- Metod *blast* se koristi za dokazivanje kompleksnih logičkih tvrđenja. Naglasimo još jednom da se u Isabelle/HOL poziv funkcije piše bez zagrade, tj. umesto $f(x)$ koristi se $f\ x$.

lemma $\neg\ (\forall\ x.\ P\ x \wedge Q\ x) \longleftrightarrow (\exists\ x.\ \neg\ P\ x\ \vee \neg\ Q\ x)$
by *blast*

Napomena: Ako trenutno ne znamo da dokažemo neku lemu, ili želimo njeno dokazivanje da ostavimo za kasnije i da nastavimo sa razvijanjem tekuće teorije možemo koristiti naredbe *sorry* i *oops*. Da bi ove dve naredbe mogle da se koriste, potrebno je uključiti *quick and dirty* mod naredbom: `declare [[quick_and_dirty = true]]`

- Naredbu *sorry* koristimo da bismo odustali od dokaza, ova lema će biti registrovana u okviru Isabelle/HOL sistema, može biti korišćena u narednim dokazima, i očekuje se da je dokažemo kasnije.
- Naredbu *oops* koristimo kada želimo da odustanemo od dokaza leme, ali ne želimo ni da bude registrovana u našoj tekućoj teoriji.

declare [[*quick-and-dirty* = *true*]]

lemma $(A \wedge B) \longrightarrow (B \vee A)$
sorry — dokaz ove leme smo ostavili za kasnije

lemma $A \longrightarrow A \wedge B$
oops — odustali smo i ova lema nije registrovana u sistemu, a pritom nije ni tačna

Upotreba ovih naredbi je dozvoljena prilikom pripreme teorije koju razvijamo. Pre nego što možemo smatrati da je tekuća celokupna teorija dokazana, moramo proći kroz sve dokaze i osloboditi se upotrebe *sorry* i *oops* naredbi.

2.5.6 Sledgehammer

U situacijama kada korisnik ne želi da raspisuje detaljan dokaz ili ne može da napiše ispravan dokaz, a ni jedan od ponuđenih automatskih alata ne uspeva da dokažu tekući cilj, na raspolaganju ima još jedan veoma moćan alat.

Komanda `sledgehammer` poziva nekoliko eksternih automatskih dokazivača teorema (*ATPs - Automated Theorem Provers*) koji se pozivaju sa vremenskim ograničenjem od podrazumevanih 30 sekundi u pokušaju da dokažu tekući cilj. Neki dokazivači su implementirani kao deo Isabelle/HOL sistema, dok se drugi pozivaju preko interneta. Najveća prednost ove komande je u tome što se u toku pretrage za dokazom koristi kompletna biblioteka lema (i korisničkih lema i onih koji pripadaju teorijama koje su uključene naredbom *imports*), pa nije potrebno eksplicitno navoditi koje leme želimo da koristimo u dokazu. Ova naredba će biti često korišćena u kasnijim poglavljima, pogotovo u drugom delu knjige koji je posvećen programiranju.

2.6 Matematička tvrđenja u Isabelle/HOL

U ovom poglavlju će biti ilustrovano zapisivanje nekoliko tipičnih tvrđenja koja se javljaju u matematičkoj logici. Tvrđenja su preuzeta iz različitih izvora da bismo ilustrovali izazove prilikom zapisivanja matematičkog teksta u okviru formalnog jezika koji koristi Isabelle/HOL. Prikazaćemo primere preuzete iz jednog udžbenika Matematičkog fakulteta, primere preuzete sa Wikipedije i iz knjige za matematičku logiku.

Prilikom dokazivanja ovih tvrđenja koristićemo samo automatski alat *auto*.

2.6.1 Formule logike prvog reda

Naredni primeri su preuzeti iz knjige Veštačka inteligencija, profesora Predraga Janičića i Mladena Nikolića http://poincare.matf.bg.ac.rs/~janicic/courses/vi.pdf. Kako je sama knjiga iz oblasti logike, za očekivati je da ovi primeri budu najjednostavniji za zapisivanje. Sami logički veznici i kvantifikatori u Isabelle/HOL se zapisuju na isti način kao u matematičkim udžbenicima, pa su ove formule identične onim koje bi se našle u rešenju iz ovog udžbenika.

Zadatak 2.1. Zapisati sledeću rečenicu u logici prvog reda. Potom je dokazati automatskim dokazivačima:

"Ako onaj ko laže taj i krade i ako bar neko laže, onda neko i krade."

lemma $((\forall\ x.\ laze\ x \longrightarrow krade\ x) \wedge (\exists\ x.\ laze\ x)) \longrightarrow (\exists\ x.\ krade\ x)$
by *auto*

Zadatak 2.2. Zapisati sledeće tvrđenje u logici prvog reda. Potom ga dokazati automatskim dokazivačima.
Ako "ko radi taj ima ili troši" i "ko ima taj peva" i "ko troši taj peva", onda "ko radi taj peva".

lemma $((\forall\ x.\ radi\ x \longrightarrow ima\ x \vee trosi\ x)$
$\wedge\ (\forall\ x.\ ima\ x \longrightarrow peva\ x)$
$\wedge\ (\forall\ x.\ trosi\ x \longrightarrow peva\ x))$
$\longrightarrow\ (\forall\ x.\ radi\ x \longrightarrow peva\ x)$
by *auto*

Zadatak 2.3. Dokazati sledeću formulu:
$(\forall\ x)(p(x) \Rightarrow q(x)) \Rightarrow (\forall\ x)(p(x) \Rightarrow (q(x) \wedge p(x)))$.

lemma $(\forall\ x.\ p\ x \longrightarrow q\ x) \longrightarrow (\forall\ x.\ p\ x \longrightarrow (q\ x \wedge p\ x))$
by *auto*

Zadatak 2.4. Zapisati konjunkciju sledećih rečenica kao formulu logike prvog reda i dokazati da je ona nezadovoljiva:

- Ako je X prijatelj osobe Y, onda je i Y prijatelj osobe X i
- Ako je X prijatelj osobe Y, onda X voli Y i
- Ne postoji neko ko je povredio osobu koju voli i
- Osoba Y je povredila svog prijatelja X

  Napomena: Kada dokazujemo da je formula nezadovoljiva, to znači da treba da pokažemo da se iz njenih pretpostavki može izvesti kontradikcija odnosno logička konstanta *False*.

lemma $((\forall\ x.\ \forall\ y.\ prijatelj\ x\ y \longrightarrow prijatelj\ y\ x)$
$\wedge\ (\forall\ x.\ \forall\ y.\ prijatelj\ x\ y \longrightarrow voli\ x\ y)$
$\wedge\ \neg\ (\exists\ x.\ \exists\ y.\ voli\ x\ y \wedge povredio\ x\ y)$
$\wedge\ (\exists\ y.\ \exists\ x.\ prijatelj\ y\ x \wedge povredio\ y\ x))$
$\longrightarrow\ False$
by *blast*

Zadatak 2.5. Zapisati u logici prvog reda naredno tvrđenje. Potom ga dokazati automatskim dokazivačima.

Ako "šta leti to ima krila i lagano je" i "šta pliva, to nema krila", onda "šta pliva, to ne leti".

lemma $((\forall\ x.\ leti\ x \longrightarrow krila\ x \wedge lagano\ x)$
$\quad \wedge\ (\forall\ x.\ pliva\ x \longrightarrow \neg\ krila\ x))$
$\quad \longrightarrow (\forall\ x.\ pliva\ x \longrightarrow \neg\ leti\ x)$
by *auto*

Zadatak 2.6. Na jeziku logike prvog reda zapisati i dokazati automatskim dokazivačima da je sledeća rečenica valjana:

"Ako postoji cipela koja u svakom trenutku odgovara svakoj nozi, onda za svaku nogu postoji cipela koja joj u nekom trenutku odgovara i za svaku nogu postoji trenutak takav da postoji cipela koja joj u tom trenutku odgovara".

lemma $(\exists\ x.\ \forall\ t.\ \forall\ y.\ cipela\ x \wedge odgovara\ x\ y\ t) \longrightarrow$
$\quad ((\forall\ y.\ \exists\ x.\ \exists\ t.\ odgovara\ x\ y\ t) \wedge (\forall\ y.\ \exists\ t.\ \exists\ x.\ odgovara\ x\ y\ t))$
by *auto*

Zadatak 2.7. U logici prvog reda zapisati rečenicu "svaka dva čoveka se vole ili ne vole" i dokazati da je dobijena formula valjana.

lemma $\forall\ x.\ \forall\ y.\ voli\ x\ y \vee \neg\ voli\ x\ y$
by *auto*

Zadatak 2.8. U logici prvog reda pokazati da je rečenica "ko rano rani, ceo dan je pospan" logička posledica rečenica "ko rano rani ceo dan je pospan ili dve sreće grabi" i "ko dve sreće grabi, ceo dan je pospan".

lemma $(\forall\ x.\ rani\ x \longrightarrow pospan\ x \vee dve\text{-}srece\ x) \wedge$
$\quad (\forall\ x.\ dve\text{-}srece\ x \longrightarrow pospan\ x) \longrightarrow$
$\quad (\forall\ x.\ rani\ x \longrightarrow pospan\ x)$
by *auto*

Zadatak 2.9. Pokazati da iz tvrđenja "dve nemimoilazne prave se seku ili su paralelne", "prave koje se seku pripadaju istoj ravni" i "prave koje su paralelne pripadaju istoj ravni" sledi tvrđenje "dve nemimoilazne prave pripadaju istoj ravni".

lemma ($\forall$ *x*. $\forall$ *y*. *nemimoilazne x y* $\longrightarrow$ *seku x y* $\vee$ *paralelne x y*)
$\wedge$ ($\forall$ *x*. $\forall$ *y*. *seku x y* $\longrightarrow$ *ista-ravan x y*)
$\wedge$ ($\forall$ *x*. $\forall$ *y*. *paralelne x y* $\longrightarrow$ *ista-ravan x y*)
$\longrightarrow$ ($\forall$ *x*. $\forall$ *y*. *nemimoilazne x y* $\longrightarrow$ *ista-ravan x y*)
by *blast*

Zadatak 2.10. Dokazati da je rečenica "Janko ruča kod kuće ili pere sudove u restoranu" logička posledica rečenica "Svako ruča kod kuće ili u restoranu", "ko ruča u restoranu i nema novca, taj pere sudove u restoranu" i "Janko nema novca".

lemma ($\forall$ *x*. *kuca x* $\vee$ *restoran x*) $\wedge$
($\forall$ *x*. *restoran x* $\wedge$ $\neg$ *novac x* $\longrightarrow$ *sudopera x*) $\wedge$
$\neg$ *novac Janko*
$\longrightarrow$ (*kuca Janko* $\vee$ *sudopera Janko*)
by *auto*

Zadatak 2.11. Pokazati da je rečenica "Svako dete voli da se igra." logička posledica rečenica "Svaki dečak voli da se igra", "Svaka devojčica voli da se igra", "Dete je dečak ili je devojčica."

lemma ($\forall$ *x*. *decak x* $\longrightarrow$ *igra x*)
$\wedge$ ($\forall$ *x*. *devojcica x* $\longrightarrow$ *igra x*)
$\wedge$ ($\forall$ *x*. *dete x* $\longrightarrow$ *decak x* $\vee$ *devojcica x*)
$\longrightarrow$ ($\forall$ *x*. *dete x* $\longrightarrow$ *igra x*)
by *auto*

Zadatak 2.12. Zapisati sledeće rečenice u logici prvog reda i pokazati da su zajedno kontradiktorne:
Ko se vozi avionom, dosta zarađuje; Ko dosta zarađuje, puno radi; Janko se vozi avionom; Janko ne radi puno.

lemma ($\forall$ *x*. *avion x* $\longrightarrow$ *zaradjuje x*) $\wedge$ ($\forall$ *x*. *zaradjuje x* $\longrightarrow$ *radi x*) $\wedge$ (*avion Janko*) $\wedge$ ($\neg$ *radi Janko*)
$\longrightarrow$ *False*
by *auto*

Zadatak 2.13. Dokazati da je rečenica "Pera voli da pleše" logička posledica rečenica "Svako ko je srećan voli da peva", "Svako ko voli da peva, voli da pleše" i "Pera je srećan".

lemma ($\forall$ *x*. *srecan x* $\longrightarrow$ *peva x*)
$\wedge$ ($\forall$ *x*. *peva x* $\longrightarrow$ *plese x*)
$\wedge$ (*srecan Pera*)
$\longrightarrow$ (*plese Pera*)
by *auto*

Zadatak 2.14. Pokazati da ako važe sledeće rečenice: "svako ima rođaka na moru ili na planini", "ko ima rođaka na moru, bio je na moru" i "ko ima rođaka na planini, bio je na planini" ne može važiti rečenica "neko nije bio ni na moru ni na planini".

lemma ($\forall$ *x*. *rodjak-more x* $\vee$ *rodjak-planina x*)
$\wedge$ ($\forall$ *x*. *rodjak-more x* $\longrightarrow$ *letovanje x*)
$\wedge$ ($\forall$ *x*. *rodjak-planina x* $\longrightarrow$ *zimovanje x*)
$\longrightarrow$ $\neg$ ($\exists$ *x*. $\neg$ *letovanje x* $\wedge$ $\neg$ *zimovanje x*)
by *auto*

Zadatak 2.15. Na jeziku logike prvog reda zapisati sledeće rečenice i dokazati da su skupa nezadovoljive:
- "Svaka dva brata imaju zajedničkog roditelja."
- "Roditelj je stariji od deteta."
- "Postoje braća."
- "Nijedna osoba nije starija od druge.

lemma ($\forall$ *x*. $\forall$ *y*. *braca x y* $\longrightarrow$ ($\exists$ *z*. *roditelj z x* $\wedge$ *roditelj z y*)) $\wedge$
($\forall$ *x*. $\forall$ *y*. *roditelj x y* $\longrightarrow$ *stariji x y*) $\wedge$
($\exists$ *x*. $\exists$ *y*. *braca x y*) $\wedge$
$\neg$ ($\exists$ *x*. $\exists$ *y*. *stariji x y*) $\longrightarrow$
False
by *auto*

Možemo koristiti i drugi način zapisivanja, uz pomoć spoljašnje sintakse:

lemma
assumes $\forall$ *x y*. *braca x y* $\longrightarrow$ ($\exists$ *z*. *roditelj z x* $\wedge$ *roditelj z y*)
assumes $\forall$ *x*. $\forall$ *y*. *roditelj x y* $\longrightarrow$ *stariji x y*
assumes $\exists$ *x y*. *braca x y*
assumes $\neg$ ($\exists$ *x y*. *stariji x y*)
shows *False*
using *assms*
by *auto*

2.6.2 Silogizmi

Silogizmi predstavljaju ispravne načine zaključivanja u kojima se zaključak izvodi na osnovu dve date premise. Njih je sistematično proučavao još Aristotel. Naredni primeri silogizama su preuzeti sa sajta https://en.wikipedia.org/wiki/Syllogism. Prilikom navođenja ovih primera, namerno je zadržana originalna formulacija na engleskom jeziku da bi predstavili realnu situaciju koja se dešava prilikom prevođenja.

Primeri iz ovog poglavlja će biti za nijansu teži za interpretiranje od primera iz prethodnog poglavlja. U prethodnom poglavlju smo radili sa preciznim rečenicama preuzetim iz udžbenika za matematičku logiku. Međutim, ono što možemo primetiti u primerima iz ovog poglavlja je da je za nijansu teži problem interpretacije rečenica koje su predstavljene *običnim* tekstom (naspram teksta koji se koristi u matematičkim udžbenicima).

Prvo ćemo pogledati jedan jednostavni primer čije rešenje će biti prikazano uz pomoć logičke formule, pa nakon toga uz pomoć *assumes-shows* bloka.

Zadatak 2.16. Zapisati tvđenje predstavljeno sledećim rečenicama:

Barbara (AAA-1)

All men are mortal. (MaP)

All Greeks are men. (SaM)

— All Greeks are mortal. (SaP)

lemma $(\forall\ x.\ men\ x \longrightarrow mortal\ x) \wedge (\forall\ x.\ greek\ x \longrightarrow men\ x) \longrightarrow$
$(\forall\ x.\ greek\ x \longrightarrow mortal\ x)$
by *auto*

Isto ovo tvrđenje može biti zapisano uz pomoć *assumes-shows* bloka na sledeći način. Isto kao i ranije, prilikom korišćenja ključne reči *assumes*, pretpostavke moraju biti eksplicitno uključene u dokaz navođenjem ključnih reči *using assms*.

lemma
assumes $\forall\ x.\ men\ x \longrightarrow mortal\ x$
and $\forall\ x.\ greek\ x \longrightarrow men\ x$
shows $\forall\ x.\ greek\ x \longrightarrow mortal\ x$
using *assms*
by *auto*

Česta greška koja se javlja je pogrešno razumevanje i interpretiranje nekih rečenica koje se javljaju u početnom tvrđenju. Interaktivni dokazivač teorema može da kontroliše da li su dokazi izvedeni korektno, pa i da automatski pronađe dokaze jednostavnih tvrđenja. On može da ukaže na

greške u zapisu tvrđenja, a to što neki dokaz ne može da se pronađe i proveri može da sugeriše da je samo tvrđenje netačno (tj. da ne predstavlja valjanu formulu). Međutim, sistem ne proverava da li formalno zapisano tvrđenje odgovara polaznom, neformalnom tvrđenju zadatom na prirodnom jeziku, pa treba biti prilično obazriv tokom kodiranja tvrđenja datih govornim jezikom tj. tokom njihovog zapisivanja u obliku logičkih formula.

Razmotrimo dva problema koji nastaju tokom zapisivanja i dokazivanja narednog jednostavnog primera.

Zadatak 2.17. Zapisati tvđenje predstavljeno sledećim rečenicama:

Barbari (AAI-1)

All men are mortal. (MaP)

All Greeks are men. (SaM)

— Some Greeks are mortal. (SiP)

Prvi problem je primer pogrešnog zapisivanja formule koji ne može biti otkriven uz pomoć Isabelle/HOL.

Problem koji se često javlja u praksi je pogrešno zapisivanje rečenice "All men are mortal".

Ispravno zapisivanje ove rečenice uz pomoć logike prvog reda je: $(\forall\ x.\ men\ x \longrightarrow mortal\ x)$.

Neispravno zapisivanje ove formule je $(\forall\ x.\ men\ x \wedge mortal\ x)$.

U suštini, ovde imamo situaciju kada je upotrebljena konjunkcija umesto implikacije. To se ne sme mešati! Ako bi koristili formulu $(\forall\ x.\ men\ x \wedge mortal\ x)$, ona ima sledeće značenje: za svaki objekat x (kakav god on bio) važi uvek *men* x i *mortal* x (odnosno važi za svaki objekat x i da je x *čovek* i da je x *smrtan*) — što nije uvek tačno, x ne mora uopšte biti ljudsko biće pa prema tome ovaj zapis nije korektan.

Generalni savet prilikom zapisivanja formule na osnovu teksta, bi bio da nakon zapisivanja formule pokušamo da tu formulu pročitamo iznova (nezavisno) kao rečenicu na srpskom jeziku i pogledamo da li dobijamo značenje od koga smo krenuli.

Drugi problem (koji se takođe može primetiti u narednom primeru) je greška koju može otkriti Isabelle/HOL, pri čemu se radi o nepotpunosti informacija koje su date neformalnim rečenicama.

Pokušajmo da formalno zapišemo ovaj silogizam.

lemma *nepotpuno*:
$(\forall\ x.\ men\ x \longrightarrow mortal\ x) \wedge (\forall\ x.\ greek\ x \longrightarrow men\ x) \longrightarrow$
$(\exists\ x.\ greek\ x \wedge mortal\ x)$

sorry
— Auto Quickcheck found a counterexample — prazan skup pa mora da se doda pretpostavka da postoji makar jedan grk

Nakon ovakvog zapisa formule dobićemo poruku od Isabelle/HOL sistema: *Auto Quickcheck found a counterexample*: *men* = {}, *mortal* = {}, *greek* = {}. To znači da navedeno tvrđenje nije tačno (formula nije valjana), preciznije da je nađen kontraprimer u slučaju kada su sva tri navedena skupa prazna. Ako pogledamo detaljnije tvrđenje koje pokušavamo da dokažemo, znamo da važi da su svi ljudi smrtni, kao i da su svi Grci ljudi. Međutim nemamo informaciju da li Grci uopšte postoje, što je sistem uspeo da otkrije. Tako da u ovom slučaju moramo dodati još jednu pretpostavku - postoji makar jedan Grk!

Sada ćemo pogledati formulacije i rešenja nekoliko sličnih silogizama:

lemma $(\forall\ x.\ men\ x \longrightarrow mortal\ x) \wedge (\forall\ x.\ greek\ x \longrightarrow men\ x) \wedge (\exists\ x.\ greek\ x) \longrightarrow$
$(\exists\ x.\ greek\ x \wedge mortal\ x)$
by *auto*

Zadatak 2.18. Zapisati tvđenje predstavljeno sledećim rečenicama:
Celarent (EAE-1)
Similar: Cesare (EAE-2)
No reptiles have fur. (MeP)
All snakes are reptiles. (SaM)
— No snakes have fur. (SeP)

lemma $\neg\ (\exists\ x.\ reptile\ x \wedge fur\ x) \wedge (\forall\ x.\ snake\ x \longrightarrow reptile\ x) \longrightarrow$
$\neg\ (\exists\ x.\ snake\ x \wedge fur\ x)$
by *auto*

Zadatak 2.19. Zapisati tvđenje predstavljeno sledećim rečenicama:
Darii (AII-1)
Similar: Datisi (AII-3)
All rabbits have fur. (MaP)
Some pets are rabbits. (SiM)
— Some pets have fur. (SiP)

lemma $(\forall\ x.\ rabbit\ x \longrightarrow fur\ x) \wedge (\exists\ x.\ pet\ x \wedge rabbit\ x) \longrightarrow$
$(\exists\ x.\ pet\ x \wedge fur\ x)$
by *auto*

Zadatak 2.20. Zapisati tvđenje predstavljeno sledećim rečenicama:
Ferioque (EIO-1)
Similar: Festino (EIO-2), Ferison (EIO-3), Fresison (EIO-4)
No homework is fun. (MeP)
Some reading is homework. (SiM)
— Some reading is not fun. (SoP)

lemma $\neg\ (\exists\ x.\ homework\ x \longrightarrow fun\ x) \wedge (\exists\ x.\ reading\ x \wedge homework\ x) \longrightarrow$
$(\exists\ x.\ reading\ x \wedge \neg\ fun\ x)$
by *auto*

Zadatak 2.21. Zapisati tvđenje predstavljeno sledećim rečenicama:
Baroco (AOO-2)
All informative things are useful. (PaM)
Some websites are not useful. (SoM)
— Some websites are not informative. (SoP)

lemma $(\forall\ x.\ informative\ x \longrightarrow useful\ x) \wedge (\exists\ x.\ website\ x \wedge \neg\ useful\ x) \longrightarrow$
$(\exists\ x.\ website\ x \wedge \neg\ informative\ x)$
by *auto*

Zadatak 2.22. Zapisati tvđenje predstavljeno sledećim rečenicama:
Bocardo (OAO-3)
Some cats have no tails. (MoP)
All cats are mammals. (MaS)
— Some mammals have no tails. (SoP)

lemma $(\exists\ x.\ cat\ x \wedge \neg\ tail\ x) \wedge (\forall\ x.\ cat\ x \longrightarrow mammal\ x) \longrightarrow$
$(\exists\ x.\ mammal\ x \wedge \neg\ tail\ x)$
by *auto*

Zadatak 2.23. Zapisati tvđenje predstavljeno sledećim rečenicama:
Celaront (EAO-1)
No reptiles have fur. (MeP)
All snakes are reptiles. (SaM)
— Some snakes have no fur. (SoP)
U ovom zadatku će opet biti pronađen kontraprimer, kada ne postoji ni jedna zmija pa dodajemo pretpostaku da postoji.

lemma $\neg\ (\exists\ x.\ reptile\ x \wedge fur\ x) \wedge (\forall\ x.\ snake\ x \longrightarrow reptile\ x) \wedge (\exists\ x.\ snake\ x) \longrightarrow$
$(\exists\ x.\ snake\ x \wedge \neg\ fur\ x)$
by *auto*

Zadatak 2.24. Zapisati tvđenje predstavljeno sledećim rečenicama:
Camestros (AEO-2)
All horses have hooves. (PaM)
No humans have hooves. (SeM)
— Some humans are not horses. (SoP)
U ovom zadatku će opet biti pronađen kontraprimer, kada ne postoji ni jedan čovek pa dodajemo pretpostaku da postoji.

lemma $(\forall\ x.\ horse\ x \longrightarrow hoove\ x) \wedge \neg\ (\exists\ x.\ human\ x \wedge hoove\ x) \wedge (\exists\ x.\ human\ x) \longrightarrow$
$(\exists\ x.\ human\ x \wedge \neg\ horse\ x)$
by *auto*

Zadatak 2.25. Zapisati tvđenje predstavljeno sledećim rečenicama:
Felapton (EAO-3)
No flowers are animals. (MeP)
All flowers are plants. (MaS)
— Some plants are not animals. (SoP)
U ovom zadatku će opet biti pronađen kontraprimer, kada ne postoji ni jedan cvet pa dodajemo pretpostaku da postoji.

lemma $\neg\ (\exists\ x.\ flower\ x \wedge animal\ x) \wedge (\forall\ x.\ flower\ x \longrightarrow plant\ x) \wedge (\exists\ x.\ flower\ x) \longrightarrow$
$(\exists\ x.\ plant\ x \wedge \neg\ animal\ x)$
by *auto*

Zadatak 2.26. Zapisati tvđenje predstavljeno sledećim rečenicama:
Darapti (AAI-3)
All squares are rectangles. (MaP)
All squares are rhombuses. (MaS)
— Some rhombuses are rectangles. (SiP)
U ovom zadatku će opet biti pronađen kontraprimer, kada ne postoji ni jedan kvadrat pa dodajemo pretpostaku da postoji.

lemma ($\forall$ *x*. *square x* $\longrightarrow$ *rectangle x*) $\wedge$ ($\forall$ *x*. *square x* $\longrightarrow$ *rhomb x*) $\wedge$ ($\exists$ *x*. *square x*) $\longrightarrow$
($\exists$ *x*. *rhomb x* $\wedge$ *rectangle x*)
by *auto*

2.6.3 Logički lavirinti

Sledeći primeri su preuzeti iz knjige Raymond M. Smullyan, Logical Labyrinths http://logic-books.info/sites/default/files/logical_labirints.pdf

Najizazovniji problemi za zapisivanje formula u Isabelle/HOL su svakako problemi koji su izazovni i u standardnoj matematici. Ova knjiga ima jako puno primera, ali u ovom poglavlju ćemo obraditi samo početnih par.

U primerima se pominje antropolog Edgar Aberkombi koji je posetio ostrvo na kome žive vitezovi i podanici. Pri čemu zna da vitezovi uvek govore istinu a da podanici uvek lažu. I znamo da svaki stanovnik ostrva mora biti ili vitez ili podanik.

Kodiranje informacija (jedna osoba):

- Pošto znamo da vitezovi uvek govore istinu a da podanici lažu uvešćemo naredna dva kodiranja:

  kA - ovako ćemo zapisati pretpostavku da je *A zapravo vitez* (*knight*)

  $\neg$ *kA* - ovako ćemo zapisati pretpostavku da je *A zapravo podanik* (*not knight*)

- U slučaju da osoba A izjavi tvrđenje B, s obzirom da vitezovi uvek govore istinu važiće sledeće: ako je osoba A vitez onda je B tačno, odnosno ako je osoba A podanik onda je B netačno. Odnosno, važiće da je A vitez ako i samo ako je tvrđenje B tačno. Otuda ćemo uvesti naredno kodiranje za rečenicu *osoba A je izjavila tvrđenje B.*

  kA $\longleftrightarrow$ *B* - ovako ćemo zapisati *osoba A je izjavila tvrđenje B*

- Ako je činjenica B dobijena kao odgovor na pitanje Q, onda odgovor može biti yes ili no, pa je B u stvari jednako (*Q* $\longleftrightarrow$ *yes*). I zato prevodimo:

  kA $\longleftrightarrow$ (*Q* $\longleftrightarrow$ *yes*) - ovako ćemo zapisati *osoba A je dala yes/no odgovor na pitanje Q*

Ako je postavljeno pitanje Q = Da li si ti vitez?, kakav odgovor očekujemo? Na osnovu malopre opisanog prevođenja, Q = k (knight), pa odgovor

možemo dobiti formulom: $k \longleftrightarrow (k \longleftrightarrow yes)$. Imamo samo dve mogućnosti, k može biti vitez ili podanik. Ako je k vitez on će dati odgovor yes, a ako je k podanik on će lagati pa će takođe dati odgovor yes. Otuda možemo formulisati prvu lemu.

Zadatak 2.27. Svaka osoba će odgovoriti potvrdno na pitanje: *Da li si ti vitez?*.

lemma $(k \longleftrightarrow (k \longleftrightarrow yes)) \longrightarrow yes$
by *auto*

Ovakva pitanja se ređe postavljaju na ovaj način, čitljivije je kada se koristi assumes i shows. Ali tada moramo uključiti pretpostavke eksplicitno u dokaz — using assms pogledajte na dnu ekrana *Output* prozor da biste videli razliku.

lemma *no-one-admits-knave*:
assumes $k \longleftrightarrow (k \longleftrightarrow yes)$
shows *yes*
using *assms*
by *auto*

Kodiranje informacija (više osoba): Kada imamo tri osobe, svaka od njih može biti vitez (što će biti označeno sa kA) ili podanik (što će biti označeno sa $\neg kA$), pa ćemo koristiti posebno kA, kB, kC, za pretpostavke o plemstvu odnosno $yesA$, $yesB$, $yesC$, za pretpostavke o odgovorima (po potrebi).

Zadatak 2.28 (Problem 1.1.)**.** Aberkombi je razgovarao sa tri stanovnika ostrva, označimo ih sa A, B i C. Pitao je stanovnika A: "Da li si ti vitez ili podanik?" A je odgovorio ali nerazgovetno pa je Aberkombi pitao stanovnika B: "Šta je A rekao?" B je odgovorio: "Rekao je da je on podanik." Tada se uključila i osoba C i rekla: "Ne veruj mu, on laže!" Da li je osoba C vitez ili podanik?

U ovoj situaciji olakšavajuća okolnost je to što je već dokazana lema koja tvrdi da će svaki stanovnik ovog ostrva dati potvrdan odgovor na pitanje da li je on vitez.

U ovakvim zadacima, kada imamo situaciju da pored zapisivanja formalnog tvrđenja imamo i zadatak da otkrijemo kakav je logički zaključak datih pretpostavki možemo koristiti Isabelle/HOL kao pomoćnika. Naime, u ovakvim situacijama je uobičajeno da matematičar na neki način smisli ili

dokaže da je određen zaključak ispravan. Međutim, kada se ovakvi problemi formulisu u okviru Isabelle/HOL možemo iskoristiti njegovu moć da automatski proverimo koji zaključak je ispravan. Naime u ovom konkretnom primeru imamo dve mogućnosti, C je ili vitez ili podanik i Isabelle/HOL će uspešno dokazati činjenicu da C jeste vitez (što neće biti slučaj ako bi pokušali da dokažemo da je C podanik - pokušajte).

lemma *Smullyan-1-1*:
$(kA \longleftrightarrow (kA \longleftrightarrow yesA)) \wedge$
$(kB \longleftrightarrow \neg\ yesA) \wedge$
$(kC \longleftrightarrow \neg\ kB) \longrightarrow$
kC
by *auto*

lemma *Smullyan-1-1′*:
assumes $kA \longleftrightarrow (kA \longleftrightarrow yesA)$
assumes $kB \longleftrightarrow \neg\ yesA$
assumes $kC \longleftrightarrow \neg\ kB$
shows kC
using *assms*
by *auto*

Zadatak 2.29 (Problem 1.2.)**.** Aberkombi je pitao stanovnika A koliko među njima trojicom ima podanika. A je opet odgovorio nerazgovetno, tako da je Aberkombi pitao stanovnika B šta je A rekao. B je rekao da je A rekao da su tačno dvojica podanici. Ponovo je stanovnik C tvrdio da B laže. Da li je u ovoj situaciji moguće odrediti da li je C vitez ili podanik?

U ovom slučaju potrebno je da definišemo novi predikat koji će nam omogućiti da zapišemo traženo tvrđenje. Predikat uvodimo ključnom rečju `definition`. Međutim, kada koristimo definicije u lemama, moramo voditi računa da ih automatski alati ne koriste automatski i da sam korisnik mora eksplicitno navesti kada želi da koristi neku definiciju. To se radi pomoću naredbe: `auto simp add: exactly_two_def`.

definition *exactly-two* **where**
$exactly\text{-}two\ A\ B\ C \longleftrightarrow ((A \wedge B) \vee (A \wedge C) \vee (B \wedge C)) \wedge \neg\ (A \wedge B \wedge C)$

lemma *Smullyan-1-2*:
$(kA \longleftrightarrow (exactly\text{-}two\ (\neg\ kA)\ (\neg\ kB)\ (\neg\ kC) \longleftrightarrow yesA)) \wedge$
$(kB \longleftrightarrow yesA) \wedge$
$(kC \longleftrightarrow \neg\ kB) \longrightarrow$
kC

by (*auto simp add*: *exactly-two-def*)

lemma *Smullyan-1-2′*:
assumes $kA \longleftrightarrow (exactly\text{-}two\ (\neg\ kA)\ (\neg\ kB)\ (\neg\ kC) \longleftrightarrow yesA)$
assumes $kB \longleftrightarrow yesA$
assumes $kC \longleftrightarrow \neg\ kB$
shows kC
using *assms*
by (*auto simp add*: *exactly-two-def*)

Zadatak 2.30. Da li se zaključak prethodnog tvrđenja menja ako B promeni svoj odgovor i kaže da je A rekao da su tačno dva od njih vitezovi?

Odgovor je *Da*. Zaključak se menja i sada glasi, C nije vitez odnosno C je podanik.

lemma *Smullyan-1-2″*:
$(kA \longleftrightarrow (exactly\text{-}two\ kA\ kB\ kC \longleftrightarrow yesA)) \wedge$
$(kB \longleftrightarrow yesA) \wedge$
$(kC \longleftrightarrow \neg\ kB) \longrightarrow$
$(\neg\ kC)$
by (*auto simp add*: *exactly-two-def*)

lemma *Smullyan-1-2‴*:
assumes $kA \longleftrightarrow (exactly\text{-}two\ kA\ kB\ kC \longleftrightarrow yesA)$
assumes $kB \longleftrightarrow yesA$
assumes $kC \longleftrightarrow \neg\ kB$
shows $\neg\ kC$
using *assms*
by (*auto simp add*: *exactly-two-def*)

end

2.7 Primer teorije

theory *Sledbenik*
imports *Main*
begin

definition *sledbenik* :: $nat \Rightarrow nat$ **where**
$sledbenik\ x = x + 1$

lemma $sledbenik\ (sledbenik\ x) = x + 2$

unfolding *sledbenik-def*
by *auto*

lemma
assumes $x > 0 \wedge x < 3$
shows *sledbenik* $x > 1 \wedge$ *sledbenik* $x < 4$
using *assms*
unfolding *sledbenik-def*
by *auto*

end

3

Dokazi u matematičkoj logici

3.1 Prirodna dedukcija u Isabelle/HOL

theory *Cas2-vezbe*

imports *Main*

begin

Formalizacija matematike podrazumeva zapis svih tvrđenja na preciznom, formalnom jeziku i njihovo dokazivanje u okviru određenog formalnog sistema, koji definiše pojam dokaza. Razvijeni su mnogi formalni sistemi, ali jedan od najpoznatijih i slobodno možemo reći naprirodnijih je *prirodna dedukcija*, koju je uveo Gerhard Gencen, 1935. godine []. Dokazi u prirodnoj dedukciji se izvode korišćenjem malog skupa jasno definisanih pravila (uključujući i jednu aksiomu). Ova pravila podsećaju na pravila dokazivanja koja se koriste u svakodnevnoj, neformalnoj matematici. Razlikovaćemo *prirodnu dedukciju za iskaznu logiku* i *prirodnu dedukciju za logiku prvog reda*. U zavisnosti od toga da li se među pravilima nalaze i pravila koja omogućavaju dokaze proizvoljnih tvrđenja svođenjem na kontradikciju razlikovaćemo *intuicionističku* i *klasičnu prirodnu dedukciju* (više reči o ovome biće dato u posebnom poglavlju).

Za klasičnu prirodnu dedukciju za iskaznu logiku važi sledeća teorema potpunosti i saglasnosti: *Iskazna formula je dokaziva u klasičnoj prirodnoj dedukciji ako i samo ako je tautologija.* Za klasičnu logiku prvog reda važi sledeća teorema potpunosti i saglasnosti: *Formula logike prvog reda je dokaziva u klasičnoj prirodnoj dedukciji ako i samo ako je valjana.* Podsetimo se iskazne tautologije su one iskazne formule koje su tačne u svakoj iskaznoj valuaciji, dok su valjane formule prvog reda one formule koje su tačne u svakom modelu (pri svakoj valuaciji). Naglasimo da ćemo u logici prvog reda uvek posmatrati samo *rečenice* tj. formule bez slobodnih promenljivih. Tačnost iskaznih formula se može utvrditi ispitivanjem svih iskaznih valuacija (semantički) i automatski dokazivači za iskaznu logiku su češće zasnovani na tom principu nego na korišćenju prirodne dedukcije. Sa druge strane u logici prvog reda nije moguće ispitati sve moguće domene i interpretacije date formule (njih najčešće ima beskonačno mnogo) i valjanost formula se stoga po pravilu ispituje sintaksički (korišćenjem prirodne dedukcije ili nekog srodnog formalizma).

Prirodna dedukcija predstavlja osnovu sistema Isabelle/HOL i svaki dokaz koji se u ovom sistemu proverava mora da bude sveden na pravila prirodne dedukcije. Međutim, zapis dokaza na nivou elementarnih pravila prirodne dedukcije može biti veoma komplikovan – takvi dokazi su veoma dugački i

potrebno je primeniti jako veliki broj pravila da bi se dokazalo i najelementarnije tvrđenje. Stoga sistem Isabelle/HOL uvodi veliki broj automatskih metoda (npr. *auto*, *blast*, *metis*, ...) koji korisnicima olakšavaju dokazivanje. Svi ti metodi u krajnjoj instanci proizvode dokaz u sistemu prirodne dedukcije. Međutim, taj dokaz je skriven od korisnika, ali je upravo u tom obliku dokaz je vidljiv sistemu i on se "iza scene" proverava.

Još jedan oblik "sakrivanja" pravila prirodne dedukcije od krajnjeg korisnika je korišćenje jezika za zapis dokaza Isar. Jedan od ciljeva kreiranja tog jezika bio je da formalni dokazi u sistemu Isabelle/HOL što je više moguće liče na neformalne dokaze iz matematičkih udžbenika. Iako se dokazi iz udžbenika u nekoj meri zasnivaju na pravilima prirodne dedukcije, primena tih pravila je najčešće implicitna (retko kada se u udžbeniku iz analize, geometrije, kombinatorike ili neke druge matematičke oblasti autor direktno poziva na modus ponens ili neko slično pravilo prirodne dedukcije, iako se ti dokazi suštinski zasnivaju na primeni tih pravila).

Uvođenje jezika Isar i automatskih metoda dokazivanja sa jedne, i prirodne dedukcije sa druge strane možemo slobodno uporediti sa odnosom programiranja na višim programskim jezicima i programiranjem na asembleru – iako se svaki program u krajnjoj instanci prevede i zapiše na asembleru pre svog izvršavanja, mnogo je udobnije programirati korišćenjem višeg nivoa apstrakcije. Slično, dokaze je mnogo udobnije pisati u jeziku Isar uz primenu automatskih metoda, međutim, da bi oni mogli da budu provereni, oni se automatski svode na dokaze zapisane u obliku pravila prirodne dedukcije (za razliku od programa koji se izvršavaju, dokazi se proveravaju).

Dakle, sve leme koje se javljaju u ovom poglavlju se mogu veoma jednostavno dokazati korišćenjem automatskih alata (praktično komandom *by auto*), ali te jednostavne leme će biti dokazane u sistemu prirodne dedukcije (što se u praksi retko kad koristi), da bi se prikazalo kako se grade dokazi koji koriste samo osnovna pravila prirodne dedukcije. Poznavanje pravila prirodne dedukcije može doprineti kasnijem lakšem razumevanju jezika Isar. Dodatno, izučavanje formalnog sistema kakva je prirodna dedukcija može čitaocu razjasniti koncept formalnog dokaza.

3.2 Pravila prirodne dedukcije

Svako od pravila prirodne dedukcije za iskaznu logiku vezano za neki logički veznik, dok kod logike prvog reda postoje i pravila vezana za kvantifikatore. U intuicionističkoj logici nemamo drugih pravila, dok u klasičnoj logici postoji bar još jedno dodatno pravilo, koje nije vezano ni za jedan konkre-

tan logički simbol. Za svaki logički veznik tj. kvantifikator postoje *pravila uvođenja* i *pravila eliminacije*. Ta pravila ćemo nazivati *intuicionistička pravila*.

Pravila uvođenja nekog veznika tj. kvantifikatora nam govore na koji način možemo da dokažemo tvrđenja kod kojih se taj veznik tj. kvantifikator javlja kao primarni u zaključku.

Na primer, da bi se dokazalo tvrđenje čiji je zaključak zapisan kao konjunkcija dva elementarnija tvrđenja $P \wedge Q$, dovoljno je nezavisno dokazati ta dva elementarnija tvrđenja P i Q. Videćemo da se ovaj jednostavan postupak dokazivanja u sistemu prirodne dedukcije formalizuje kroz pravilo *"uvođenje konjunkcije"*.

Pravila eliminacije nekog veznika tj. kvantifikatora nam govore kako da izmenimo skup pretpostavki kada se u pretpostavkama tvrđenja koje se dokazuje nalazi neka formula kojoj se taj veznik tj. kvantifikator javlja kao primarni (drugim rečima, kako da iz postojećih pretpostavki izvedemo neke nove, koje logički slede iz njih).

Na primer, jedno od osnovnih pravila u neformalnom dokazivanju je pravilo *modus ponens*, koje nam omogućava da iz dokazanog tvrđenja P i dokazanog tvrđenja $P \to Q$, izvedemo tj. dokažemo tvrđenje Q. Ono u formalnom sistemu prirodne dedukcije odgovara pravilu *"eliminacije implikacije"*.

Dalje, na primer, ako je dokazano tvrđenje koje je zapisano u obliku univerzalno kvantifikovane rečenice u kojoj se tvrdi da svi objekti imaju neko svojstvo, možemo slobodno smatrati da je dokazano i tvrđenje koje tvrdi da neki konkretan objekat koji razmatramo ima to svojstvo. Videćemo da se ovaj postupak dokazivanja formalizuje kroz pravilo *"eliminacije univerzalnog kvantifikatora"*.

Proizvoljno pravilo prirodne dedukcije možemo zapisati ovako (nazovimo ga pravilo R):

$$\frac{P_1 \quad \dots \quad P_n}{Q}$$

Pravila prirodne dedukcije možemo čitati i odozgo i odozdo: ako smo dokazali pretpostavke (ono iznad crte), tada možemo smatrati da smo uspešno dokazali zaključak (ono što se nalazi ispod crte). Sa druge strane možemo čitati odozdo na gore: da bi smo dokazali traženi zaključak dovoljno je da dokažemo sve navedene pretpostavke.

Delimo ih na pravila koja se odnose na zaključak tvrđenja - *pravila uvođenja* (koja se u sistemu Isabelle/HOL koriste uz metod *rule*) i pravila koja se odnose na pretpostavke tvrđenja - *pravila eliminacije* (koja se u sistemu

Isabelle/HOL koriste uz metod *erule*). Pored ova dva metoda, u situaciji kada se zaključak koji se dokazuje već nalazi u pretpostavkama, koristi se metoda *assumption*. Ova situacija odgovara primeni aksiome prirodne dedukcije.

Navedimo jedan kratak dokaz koji koristi tri osnovne metode *rule*, *erule* i *assumption*. U prvom koraku se primenjuje pravilo uvođenja implikacije u zaključak tvrđenja (*impI*), u drugom koraku se primenjuje pravilo eliminacije konjunkcije u pretpostavkama tvrđenja (*conjE*), i u trećem koraku se prepoznaje da su pretpostavka i zaključak identični (*assumption*). Primetimo da se svaki metod navodi nakon ključne reči *apply*, dok se na kraju dokaza navodi ključna reč *done*.

lemma $A \wedge B \longrightarrow A$
  apply (*rule impI*) — *Cilj postaje:* $A \wedge B \Longrightarrow A$.
  apply (*erule conjE*) — *Cilj postaje:* $[\![A; B]\!] \Longrightarrow A$.
  apply *assumption* — *Nema više podciljeva.*
  done — *Dokazana teorema.*

Alternativa je zapisivanje ovakve formule uz pomoć *assumes*−*shows* bloka. Tada upotreba pravila uvođenja implikacije nije potrebna, mora se dodati korak *using assms* i dokaz izgleda ovako:

lemma
  assumes $A \wedge B$
  shows A
  using *assms*
  apply (*rule conjE*)
  apply *assumption*
  done

3.3 Pravila uvođenja i eliminacije za iskaznu logiku u Isabelle/HOL

Iskazne formule u sistemu Isabelle/HOL se grade pomoću logičkih veznika konjunkcije ($\wedge$), disjunkcije ($\vee$), implikacije ($\longrightarrow$), ekvivalencije ($\longleftrightarrow$) i unarnog simbola negacije ($\neg$). U nastavku ćemo izložiti pravila prirodne dedukcije za iskaznu logiku (tj. za ove veznike), a u narednom poglavlju ćemo uvesti i pravila za kvantifikatore i logiku prvog odnosno višeg reda. Pravila koja se odnose na iskazne veznike su *conjI*, *conjE*, *disjI1*, *disjI2*, *disjE*, *impI*, *impE*, *iffI*, *iffE*, *notI*, *notE*.

Veznik	Pravilo uvođenja	Pravilo eliminacije
$\wedge$	conjI	conjE
$\vee$	disjI1, disjI2	disjE
$\longrightarrow$	impI	impE
$\longleftrightarrow$	iffI	iffE
$\neg$	notI	notE

U sistemu Isabelle/HOL naredbom *thm* dobijamo zapis pravila čije ime navedemo, u sintaksi meta-logike sistema Isabelle. Naredba *thm* može biti navedena u bilo kom delu teorije, čak i u sklopu nekog dokaza, pri čemu ona ne čini deo samog dokaza tj. ne utiče na stanje u kom se dokazivač nalazi. Zapis svih pravila iskazne logike dat je u nastavku (a objašnjenje jednog po jednog od ovih pravila će biti navedeno u tekstu koji sledi)

thm *conjI* — $[\![?P;\ ?Q]\!] \Longrightarrow ?P \wedge ?Q$
thm *disjI1* — $?P \Longrightarrow ?P \vee ?Q$
thm *disjI2* — $?Q \Longrightarrow ?P \vee ?Q$
thm *impI* — $(?P \Longrightarrow ?Q) \Longrightarrow ?P \longrightarrow ?Q$
thm *iffI* — $[\![?P \Longrightarrow ?Q;\ ?Q \Longrightarrow ?P]\!] \Longrightarrow ?P = ?Q$
thm *notI* — $(?P \Longrightarrow \mathit{False}) \Longrightarrow \neg\ ?P$

thm *conjE* — $[\![?P \wedge ?Q;\ [\![?P;\ ?Q]\!] \Longrightarrow ?R]\!] \Longrightarrow ?R$
thm *disjE* — $[\![?P \vee ?Q;\ ?P \Longrightarrow ?R;\ ?Q \Longrightarrow ?R]\!] \Longrightarrow ?R$
thm *impE* — $[\![?P \longrightarrow ?Q;\ ?P;\ ?Q \Longrightarrow ?R]\!] \Longrightarrow ?R$
thm *iffE* — $[\![?P = ?Q;\ [\![?P \longrightarrow ?Q;\ ?Q \longrightarrow ?P]\!] \Longrightarrow ?R]\!] \Longrightarrow ?R$
thm *notE* — $[\![\neg\ ?P;\ ?P]\!] \Longrightarrow ?R$

3.3.1 Pravila uvođenja

Pravila uvođenja nam govore kako se dokazuje tvrđenje čiji zaključak ima određen primarni veznik. Primenjuju se metodom *rule*.

Uvođenje konjunkcije: Kao što je i očekivano, ako pojedinačno možemo da dokažemo formule P i Q, onda možemo da dokažemo i formulu $P \wedge Q$. Ovo pravilo se primenjuje kada u zaključku formule koju pokušavamo da dokažemo imamo konjunkciju kao primarni veznik.

U udžbenicima matematičke logike ovo pravilo se zapisuje na sledeći način:

$$\frac{P \quad Q}{P \wedge Q}$$

U sistemu Isabelle/HOL ovo pravilo se zove *conjI* i zapisuje se na sledeći način $[\![P;\ Q]\!] \Longrightarrow P \wedge Q$. Šematske promenljive (promenljive obeležene upitnicima) koje se javljaju u ovom pravilu mogu biti zamenjene proizvoljnim formulama. Pravilo se primenjuje sa desna na levo. Zaključak tvrđenja koje se dokazuje se unifikuje sa *?P* $\wedge$ *?Q*, i to tvrđenje se zamenjuje sa dva tvrđenja čiji su zaključci redom formule *?P* i *?Q* (instancirane u odnosu na unifikator zaključka i formulu *?P* $\wedge$ *?Q*), dok se pretpostavke prepisuju iz polaznog tvrđenja.

Prikažimo efekat primene ovog pravila na narednom jednostavnom primeru. Formula nije tautologija, pa dokaz nije moguće završiti i zato je na kraju dokaza upotrebljena ključna reč *oops*. Izlaz koji se dobija kada se naredni dokaz pokrene u Isabelle-u biće prikazan u okviru komentara kao deo samog dokaza.

Kako je glavni veznik u ovoj formuli konjunkcija i kako se ta konjunkcija nalazi u zaključku tvrđenja koje se dokazuje primenjuje se pravilo *conjI*. Nakon primene naredbe *apply* (*rule conjI*), dokazivač identifikuje dva nova cilja A i B (odnosno u opštem slučaju prvi, odnosno drugi konjunkt zadat konjunkcijom u zaključku). Nakon toga prvo se dokazuje prvi cilj, i tek nakon dokazivanja prvog cilja se započinje sa dokazivanjem drugog cilja.

lemma $A \wedge B$
  apply (*rule conjI*)
— goal (2 subgoals):
— 1. A
— 2. B
  oops

Uvođenje disjunkcije: Da bismo dokazali tvrđenje oblika $P \vee Q$ dovoljno je da dokažemo P, odnosno alternativno dovoljno je da dokažemo Q. Stoga se u udžbenicima matematičke logike navode dva pravila uvođenja disjunkcije.

$$\frac{P}{P \vee Q} \qquad \frac{Q}{P \vee Q}$$

U sistemu Isabelle/HOL postoje dva pravila *disjI1*: $P \Longrightarrow P \vee Q$ i *disjI2*: $Q \Longrightarrow P \vee Q$.

Prikažimo efekat primene ovih pravila na narednom primeru (formula nije teorema, pa dokaze nije moguće završiti). Primena pravila *disjI1* identifikuje prvi disjunkt A kao jedini (novi) cilj, dok primena pravila *disjI2* identifikuje drugi disjunkt B kao jedini cilj.

lemma $A \vee B$
apply (*rule disjI1*)
— goal (1 subgoal):
— 1. A
oops

lemma $A \vee B$
apply (*rule disjI2*)
— goal (1 subgoal):
— 1. B
oops

Pravila uvođenja disjunkcije se mogu smatrati nebezbednim pravilima. Na primer, moguće je da dokazujemo teoremu u kojoj iz nekih pretpostavki sledi zaključak $P \vee Q$. Ako primenimo, na primer, pravilo *disjI1*, tada je potrebno da dokažemo tvrđenje u kome iz istih tih pretpostavki sledi zaključak P, međutim, to tvrđenje, za razliku od polaznog, ne mora uopšte više da bude tačno tj. dokazivo. Pošto se pravila prirodne dedukcije primenjuju tokom rada automatskih metoda, i tada treba biti obazriv kada se u zaključku nalazi disjunkcija, jer neki automatski metodi pokušavaju da primene i nebezbedna pravila, poput ovih za eliminaciju disjunkcije, što u nekim slučajevima dovodi do toga da automatski alat umesto da pojednostavi teoremu koju treba dokazati, svede teoremu na tvrđenje koje se više ne može dokazati.

Uvođenje implikacije: Da bismo dokazali $P \longrightarrow Q$ treba da pretpostavimo da P važi i pokušamo da dokažemo Q. Ako to uspemo, onda smo dokazali implikaciju.

U udžbenicima matematičke logike ovo pravilo se zapisuje na sledeći način:

$$\frac{\begin{array}{c}[P]\\ \vdots \\ Q\end{array}}{P \to Q}$$

U sistemu Isabelle/HOL pravilo se zove *impI* i zapisuje se na sledeći način $(P \Longrightarrow Q) \Longrightarrow P \longrightarrow Q$. Oba simbola $\Longrightarrow$ i $\longrightarrow$ označavaju implikaciju ali je prvi simbol deo meta-logike sistema Isabelle i ovde se koristi za zapisivanje pravila izvođenja. Sa druge strane veznik $\longrightarrow$ predstavlja samo jedan od veznika koji se koriste u logici višeg reda. Dok se veznik $\longrightarrow$ koristi za zapisivanje formula logike višeg reda, veznik $\Longrightarrow$ se koristi za odvajanje pretpostavki tvrđenja koje se dokazuje od njegovog zaključka.

Prikažimo efekat primene ovog pravila na narednom primeru (formula nije teorema, pa dokaz nije moguće završiti). Nakon primene pravila *impI* vidimo da je formula $A \Longrightarrow B$ identifikovana kao novi cilj, odnosno Isabelle je logičku implikaciju zamenio meta-implikacijom i odvojio je pretpostavke tvrđenja (A) od njegovog zaključka (B).

lemma $A \longrightarrow B$
apply (*rule impI*)
— goal (1 subgoal):
— 1. $A \Longrightarrow B$
oops

Uvođenje ekvivalencije: Da bismo dokazali $P \longleftrightarrow Q$ treba da dokažemo dva cilja. Prvo da (uz ostale pretpostavke) pretpostavimo da važi P i da pokušamo da dokažemo Q, i nakon toga, da (uz ostale pretpostavke) pretpostavimo da važi Q i da pokušamo da dokažemo P. U udžbenicima matematičke logike, ovo pravilo se obično ne navodi eksplicitno, već se smatra da je $P \longleftrightarrow Q$ samo skraćeni zapis za $P \longrightarrow Q \wedge Q \longrightarrow P$.

U sistemu Isabelle/HOL pravilo se zove *iffI* i zapisuje se ovako $[\![P \Longrightarrow Q; Q \Longrightarrow P]\!] \Longrightarrow P = Q$. Dakle, njime se dokaz ekvivalencije efektivno svodi na dokaz dve implikacije.

Prikažimo efekat primene ovog pravila na narednom primeru (formula nije teorema, pa dokaz nije moguće završiti). Nakon primene pravila *iffI* logička ekvivalencija u zaključku teoreme se svodi na dve implikacije $A \Longrightarrow B$ i $B \Longrightarrow A$ (primetimo da se primenom ovog pravila u formulama odmah dobija veznik meta-implikacije).

lemma $A \longleftrightarrow B$
apply (*rule iffI*)
— goal (2 subgoals):
— 1. $A \Longrightarrow B$
— 2. $B \Longrightarrow A$
oops

Uvođenje negacije: Ako iz pretpostavke P (i ostalih pretpostavki) možemo da dokažemo *False*, odnosno ako možemo da izvedemo kontradikciju pod pretpostavkom P, onda možemo da zaključimo da važi $\neg P$.

U udžbenicima matematičke logike ovo pravilo se zapisuje na sledeći način:

$$\frac{\begin{array}{c}[P]\\ \vdots \\ False\end{array}}{\neg P}$$

U sistemu Isabelle/HOL pravilo se zove *notI* i zapisuje se na sledeći način $(P \Longrightarrow False) \Longrightarrow \neg\ P$.

Primetimo da se ovim pravilom dokaz vrši svođenjem na kontradikciju, međutim, jako je važno naglasiti da se u intuicionističkoj logici na taj način mogu dokazati samo negirana tvrđenja. Dokazivanje pozitivnih tvrđenja svođenjem na kontradikciju zahteva primenu pravila klasične logike, o čemu će više reči biti u narednim poglavljima.

Prikažimo efekat primene ovog pravila na narednom primeru (formula nije teorema, pa dokaz nije moguće završiti). Nakon primene pravila *notI* novi cilj postaje formula $A \Longrightarrow False$.

lemma $\neg\ A$
  apply (*rule notI*)
— goal (1 subgoal):
— 1. $A \Longrightarrow False$
  oops

Ovde ćemo navesti nekoliko jednostavnih tvrđenja čiji se dokazi mogu izvesti primenom samo pravila uvođenja:

lemma $A \longrightarrow A \vee B$
  apply (*rule impI*)
— *1.* $A \Longrightarrow A \vee B$
  apply (*rule disjI1*) — *Alternativa disjI2 ne uspeva.*
— *1.* $A \Longrightarrow A$
  apply *assumption*
— *No subgoals*!
  done

lemma $B \longrightarrow A \vee B$
  apply (*rule impI*)
— *1.* $B \Longrightarrow A \vee B$
  apply (*rule disjI2*)
— *1.* $B \Longrightarrow B$
  apply *assumption*
  done

lemma $B \longrightarrow (A \wedge A) \vee B$

apply (*rule impI*)
— *1.* $B \Longrightarrow A \wedge A \vee B$
apply (*rule disjI2*)
— *1.* $B \Longrightarrow B$
apply *assumption*
done

lemma $A \longrightarrow (A \wedge A) \vee B$
apply (*rule impI*)
— *1.* $A \Longrightarrow A \wedge A \vee B$
apply (*rule disjI1*)
— *1.* $A \Longrightarrow A \wedge A$
apply (*rule conjI*)
— *1.* $A \Longrightarrow A$
— *2.* $A \Longrightarrow A$
apply *assumption*
— *1.* $A \Longrightarrow A$
apply *assumption*
done

lemma $A \longrightarrow (A \vee B) \wedge (B \vee A)$
apply (*rule impI*)
— *1.* $A \Longrightarrow (A \vee B) \wedge (B \vee A)$
apply (*rule conjI*)
— *1.* $A \Longrightarrow A \vee B$
— *2.* $A \Longrightarrow B \vee A$
apply (*rule disjI1*)
— *1.* $A \Longrightarrow A$
— *2.* $A \Longrightarrow B \vee A$
apply *assumption*
— *1.* $A \Longrightarrow B \vee A$
apply (*rule disjI2*)
— *1.* $A \Longrightarrow A$
apply *assumption*
done

lemma $A \longleftrightarrow A$
apply (*rule iffI*)
— *1.* $A \Longrightarrow A$
— *2.* $A \Longrightarrow A$
apply *assumption*
— *1.* $A \Longrightarrow A$
apply *assumption*

done

3.3.2 Pravila eliminacije

Pravila eliminacije nam daju nam mogućnost da pretpostavke tvrđenja koje se dokazuje u kojima se javlja određeni veznik transformišemo na određeni način i tako iskoristimo "znanje" koje je kodirano tim pretpostavkama. U sistemu Isabelle/HOL se primenjuju metodom *erule* koja pokušava da navedeno pravilo primeni neku od trenutnih pretpostavki. Ako je primena pravila uspešna, ta pretpostavka se briše i formiraju se novi ciljevi koje je potrebno dokazati (u zavisnosti od pravila, u njima i pretpostavke i zaključak mogu biti izmenjeni).

Za razliku od zaključka tvrđenja, koji je uvek jedinstven, pretpostavki može biti više. Tako se može desiti da se jedno pravilo eliminacije može da se primeni na više različitih pretpostavki. U ovakvim situacijama Isabelle/HOL uvek bira prvu pretpostavku na koju se dato pravilo može primeniti. U slučaju da je potrebno izabrati neku od narednih pretpostavki, koristi se ključna reč *back*, kojom se pronalazi naredna pretpostavka na na koju je moguće primeniti navedeno pravilo eliminacije i to pravilo se na nju primenjuje. Ako takva pretpostavka ne postoji, prijavljuje se greška.

Eliminacija konjunkcije: Kada se eliminiše konjunkcija $P \wedge Q$, u prirodnoj dedukciji kakva se opisuje u udžbenicima matematičke logike postoje dva pravila u zavisnosti od toga da li se izvodi prvi ili drugi konjunkt:

$$\frac{P \wedge Q}{P} \qquad \frac{P \wedge Q}{Q}$$

Isabelle/HOL pravilo za eliminaciju konjunkcije malo odstupa od ustaljenih pravila datih u klasičnim udžbenicima matematičke logike, u kojima postoji posebno pravilo kojim se iz konjunkcije dobija P i posebno pravilo kojim se iz konjunkcije dobija Q, i odmah iz konjunkcije daje P i Q. Pravilo se zove *conjE* i zapisuje se na sledeći način $[\![P \wedge Q; [\![P;\ Q]\!] \Longrightarrow R]\!] \Longrightarrow R$. Tumačenje ovog pravila je sledeće. Pretpostavimo da je potrebno dokazati neko tvrđenje *?R*, a da se među pretpostavkama nalazi formula oblika *?P* $\wedge$ *?Q*. Tada je dovoljno dokazati tvrđenje *?R* iz nezavisnih pretpostavki *?P* i *?Q*. Drugim rečima, ako možemo da dokažemo *?P* $\wedge$ *?Q* i ako možemo da dokažemo da pod pretpostavkama *?P* i *?Q* važi *?R* (tj. $[\![?P;\ ?Q]\!] \Longrightarrow ?R$), tada možemo smatrati da smo uspešno dokazali *?R*.

Napomena: Primetimo da su leme u ovom poglavlju formulisane sa meta-implikacijom $\Longrightarrow$, naspram do sada korišćene logičke implikacije $\longrightarrow$. Razlog za to je što korišćenje meta-implikacije omogućava korisniku da u dokazu ne koristi ni jedno pravilo uvođenja.

Prikažimo efekat primene ovog pravila na narednom primeru. Pravilo eliminacije konjunkcije eliminiše konjunkciju iz pretpostavki i umesto nje dodaje njene pojedinačne konjunkte kojima korisnik dalje može pristupiti po potrebi.

lemma $A \wedge B \Longrightarrow A$
  apply (*erule conjE*)
— *1.* $[\![A; B]\!] \Longrightarrow A$
  apply *assumption*
— *No subgoals*!
  done

Eliminacija disjunkcije: Eliminacija disjunkcije odgovara dokazu po slučajevima u klasičnim matematičkim udžbenicima.

U udžbenicima matematičke logike, ovo pravilo se zapisuje na sledeći način:

$$\frac{\begin{array}{ccc} & [P] & [Q] \\ & \vdots & \vdots \\ P \vee Q & R & R \end{array}}{R}$$

Dakle, ako nezavisno, iz svakog disjunkta možemo da izvedemo neki zaključak, onda možemo da eliminišemo disjunkciju i da je zamenimo tim zaključkom. Primetimo da se objektna disjunkcija na neki način menja konjunkcijom na meta-nivou (potrebno je da dokaz izvršimo nezavisno i za jedan i za drugi disjunkt).

U sistemu Isabelle/HOL pravilo se zove *disjE* i zapisuje se na sledeći način $[\![P \vee Q; P \Longrightarrow R; Q \Longrightarrow R]\!] \Longrightarrow R$. Primena ovog pravila rezultira grananjem (jedno tvrđenje se zamenjuje sa dva) pri čemu prva grana odgovara prvoj pretpostavci a druga grana drugoj pretpostavci. Prvo pretpostavimo P, pa iz toga dokažemo R, pa onda nezavisno pretpostavimo Q, pa iz toga dokažemo R i onda smatramo da smo dokazali R. U toku ovog procesa R se dokazuje dva puta, iz različitih skupova pretpostavki.

Napomena: zadebljane zagrade koje se obično prikazuju u Output prozoru se mogu koristiti i prilikom zapisivanja samog tvrđenja koje se dokazuje.

Prikažimo efekat primene ovog pravila na narednom primeru (formula nije teorema i ne može se dokazati). Eliminacija disjunkcije iz pretpostavke

odgovara grananju po slučajevima i generiše dva podcilja: u prvom je jedina pretpostavka tvrđenje A, a u drugom tvrđenje B.

lemma $[\![A \vee B]\!] \Longrightarrow C$
apply (*erule disjE*)
— *1.* $A \Longrightarrow C$
— *2.* $B \Longrightarrow C$
oops

Eliminacija implikacije: Ako znamo da važi implikacija $P \longrightarrow Q$, koristi se pravilo *modus ponens* za eliminaciju implikacije. U udžbenicima matematičke logike ovo pravilo se zapisuje na sledeći način:

$$\frac{P \quad P \to Q}{R}$$

Da bi se eliminisala implikacija $P \longrightarrow Q$ neophodno je da se nezavisno (iz ostalih pretpostavki) dokaže P, i da se dokaže da iz ostalih pretpostavki i iz Q važi R.

U sistemu Isabelle/HOL ovo pravilo se zove *impE* i zapisuje se na sledeći način $[\![P \longrightarrow Q;\ P;\ Q \Longrightarrow R]\!] \Longrightarrow R$.

Ovo pravilo je još jedno od nebezbednih pravila. Može se desiti da je implikacija (na koju se primenjuje pravilo) takva da se njena pretpostavka P ne može dokazati. Jedan način da se ovo dogodi je da se eliminacija implikacije greškom primeni na pogrešnu implikaciju u pretpostavkama. U toj situaciji koristićemo ključnu reč *back* da bismo naveli dokazivač da pravilo primeni na narednu drugu pretpostavku.

Prikažimo efekat primene ovog pravila na narednom primeru (tvrđenje se ne može dokazati). Eliminacija implikacije iz pretpostavke generiše dva odvojena cilja. Kako u narednom primeru nema dodatnih pretpostavki, prvi cilj postaje leva strana implikacije (samo formula A), a drugi cilj u pretpostavkama zadržava samo desnu stranu implikacije (formula B) i u zaključku ostaje originalni cilj.

lemma $[\![A \longrightarrow B]\!] \Longrightarrow C$
apply (*erule impE*)
— *1.* A
— *2.* $B \Longrightarrow C$
oops

Eliminacija ekvivalencije: Ako znamo da važi implikacija $P \longleftrightarrow Q$, koristi se pravilo za eliminaciju ekvivalencije koje se u sistemu Isabelle/HOL

zove *iffE* i zapisuje se na sledeći način $[\![P = Q; [\![P \longrightarrow Q; Q \longrightarrow P]\!] \Longrightarrow R]\!] \Longrightarrow R$. Primena ovog pravila odgovara tumačenju da je veznik ekvivalencije samo skraćenica za dve implikacije.

Prikažimo efekat primene ovog pravila na narednom primeru (tvrđenje se ne može dokazati). Eliminacija ekvivalencije iz pretpostavke generiše dve nove pretpostavke, implikacija u jednom smeru i implikacija u obrnutom smeru.

lemma $[\![A \longleftrightarrow B]\!] \Longrightarrow C$
apply (*erule iffE*)
— *1.* $[\![A \longrightarrow B; B \longrightarrow A]\!] \Longrightarrow C$
oops

Eliminacija negacije: Iz kontradiktornih pretpostavki mozemo da zaključimo bilo šta. U udžbenicima matematičke logike ovo pravilo se primenjuje na sledeći način:

$$\frac{P \quad \neg P}{False}$$

U sistemu Isabelle/HOL ovo pravilo se zove *notE* i zapisuje se na sledeći način: $[\![\neg\ P;\ P]\!] \Longrightarrow R$. Primena ovog pravila teče tako što se u pretpostavkama pronađe neka negirana formula $\neg\ P$ i onda se iz ostalih pretpostavki pokušava dokazati formula P. Primetimo da se zaključak polaznog tvrđenja potpuno ignoriše i nakon primene pravila *notE* on nestaje iz transformisanog tvrđenja.

I ovo pravilo je nebezbedno i može se desiti da primena ovog pravila dovede do cilja koji ne možemo da dokažemo. Do toga može doći i kada u pretpostavkama tvrđenja koje se dokazuje imamo više negacija, i tada je potrebno voditi računa koju negaciju treba eliminisati. Da bismo precizno izabrali odgovarajuću negaciju, možemo koristiti naredbu *back*.

Prikažimo efekat primene ovog pravila na narednom primeru (tvrđenje se ne može dokazati). Eliminacija negacije iz pretpostavke generiše novi cilj koji kao zaključak ima pozitivan oblik formule koja stoji iza veznika $\neg$. Kako u ovom slučaju nema ostalih pretpostavki, tvrđenje se sastoji samo od tog pozitivnog oblika.

lemma $\neg\ A \Longrightarrow B$
apply (*erule notE*)
— *1.* A
oops

Bezbedna i nebezbedna pravila: Može se primetiti da primena određenih pravila može narušiti dokazivost tvrđenja. Usled toga razlikujemo *bezbedna* i *nebezbedna* pravila. *Bezbedna pravila* su *conjI*, *impI*, *iffI*, *notI*, *conjE*, *disjE* i *iffE*. *Nebezbedna pravila* su *disjI1*, *disjI2*, *impE* i *notE*. U dokazima se preporučuje da se prvo iscrpno primene sva bezbedna pravila, pa tek onda nebezbedna pravila. Prilikom primene nebezbednih pravila eliminacije treba voditi računa o tome da li su zaista primenjena na željene pretpostavke.

3.4 Primeri dokaza u prirodnoj dedukciji

U nastavku ćemo prikazati primere dokaza prirodnom dedukcijom u sistemu Isabelle/HOL. Svi oni su dati u obliku niza primena pravila navedenih komandom *apply*. Takvi dokazi se nazivaju nestrukturirani dokazi i oni nisu jednostavno čitljivi. Da bismo čitaocu olakšali praćenje navešćemo dodatna objašnjenja u okviru samih dokaza u vidu komentara (tekst koji se pojavljuje iza dugačke crtice (—)). Indentacija (uvlačenje) komandi *apply* u ovim dokazima se javlja kada primena pravila poveća broj tvrđenja (ciljeva) koje treba pokazati i tada se početak reda pomera za jedno mesto udesno (jedan razmak) u odnosnu na prethodni red. Kada se podcilj uspešno dokaže, indentacija se smanjuje za jedno mesto.

Da bi čitalac mogao da prati ove dokaze i u Isabelle/HOL i van njega, u prvih nekoliko primera biće prikazan prvo kompletan dokaz, pa onda isti taj dokaz detaljno iskomentarisan i dopunjen odgovarajućim izlazom koji se dobija u Isabelle/HOL. Komentari će biti navedeni za svaku naredbu u prvih nekoliko primera, a u kasnijim primerima komentarisaćemo samo ključne korake.

Zadatak 3.1. $A \wedge B \longrightarrow B \wedge A$

Dokaz teoreme:

lemma $A \wedge B \longrightarrow B \wedge A$
apply (*rule impI*)
apply (*erule conjE*)
apply (*rule conjI*)
apply *assumption*
apply *assumption*
done

Detaljno objašnjen dokaz teoreme:

lemma $A \wedge B \longrightarrow B \wedge A$
— Cilj je identifikovan sa: $A \wedge B \longrightarrow B \wedge A$, što znači da dokazujemo ovu iskaznu formulu iz praznog skupa pretpostavki
— Logičku implikaciju $\longrightarrow$ u zaključku možemo transformisati je u meta-implikaciju $\Longrightarrow$, korišćenjem pravila impI
apply (*rule impI*)
— Sada je cilj postao: $A \wedge B \Longrightarrow B \wedge A$, što znači da iz pretpostavke $A \wedge B$ dokazujemo zaključak $B \wedge A$
— Prvo primenjujemo pravilo za eliminaciju konjunkcije u pretpostavkama, da bismo mogli da koristimo pojedinačno pretpostavke A i B.
apply (*erule conjE*)
— Sada je cilj postao: $[\![A;\ B]\!] \Longrightarrow B \wedge A$, što znači da iz pretpostavki A i B dokazujemo zaključak $B \wedge A$
— Sada želimo da uvedemo konjunkciju u zaključku i primenjujemo pravilo *conjI* za uvođenje konjunkcije ($B \wedge A$ dokazujemo tako što nezavisno dokažemo B i A
apply (*rule conjI*)
— *Sada dobijamo dva cilja:.*
— *1.* $[\![A;\ B]\!] \Longrightarrow B$.
— *2.* $[\![A;\ B]\!] \Longrightarrow A$.
— I dokazujemo prvi cilj: $[\![A;\ B]\!] \Longrightarrow B$. Pošto je sada u pretpostavkama već prisutan trenutni zaključak, pozivamo metodu *assumption*
apply *assumption*
— Sada nestaje prvi cilj i ostaje samo drugi cilj: $[\![A;\ B]\!] \Longrightarrow A$, za koji opet možemo primeniti metodu *assumption* zato što je u pretpostavkama prisutan tekući cilj
apply *assumption*
— Nakon ovoga dobijamo izlaz: *No subgoals*! i možemo završiti tekući dokaz naredbom *done*.
done
— Nakon ove naredbe, teorema je dokazana i evidentirana u Isabelle/HOL.

Zadatak 3.2. $A \vee B \longrightarrow B \vee A$

Dokaz teoreme:

lemma $A \vee B \longrightarrow B \vee A$
apply (*rule impI*)
apply (*erule disjE*)
apply (*rule disjI2*)
apply *assumption*
apply (*rule disjI1*)
apply *assumption*
done

Detaljno objašnjen dokaz teoreme:

lemma $A \vee B \longrightarrow B \vee A$
— *Cilj je identifikovan sa:* $A \vee B \longrightarrow B \vee A$
— *Logičku implikaciju transformišemo u meta-implikaciju pravilom impI*
apply (*rule impI*)
— *Cilj je sada:* $A \vee B \Longrightarrow B \vee A$.
— *U pretpostavci imamo disjunkciju pa je eliminišemo pravilom disjE.*
apply (*erule disjE*)
— *Sada dobijamo dva cilja:.*
— *1.* $A \Longrightarrow B \vee A$.
— *2.* $B \Longrightarrow B \vee A$.
— *Prvo rešavamo prvi cilj:* $A \Longrightarrow B \vee A$. *Primećujemo da u zaključku prvog cilja imamo pretpostavku A kao drugi disjunkt pa primenjujemo pravilo disjI2 da bismo upravo taj disjunkt izdvojili.*
apply (*rule disjI2*)
— *Tekući cilj sada postaje:* $A \Longrightarrow A$ *(dok drugi cilj ostaje nepromenjen i čeka svoj red).*
— *Kada se zaključak nalazi među pretpostavkama primenjujemo naredbu assumption, čime se završava dokaz prvog podcilja.*
apply *assumption*
— *Sada rešavamo drugi podcilj:* $B \Longrightarrow B \vee A$. *Primećujemo da u zaključku imamo pretpostavku B kao prvi disjunkt pa primenjujemo pravilo disjI1 da bismo upravo taj disjunkt izdvojili.*
apply (*rule disjI1*)
— *Tekući cilj sada postaje:* $B \Longrightarrow B$.
— *Kada se zaključak nalazi među pretpostavkama primenjujemo naredbu assumption, čime se završava dokaz drugog podcilja.*
apply *assumption*
— *Dokazali smo oba cilja i dobijamo izlaz No subgoals*! *i možemo završiti tekući dokaz naredbom done.*
done
— *Nakon ove naredbe, teorema je dokazana i evidentirana u Isabelle/HOL.*

Zadatak 3.3. $A \wedge B \longrightarrow A \vee B$

Dokaz teoreme:

lemma $A \wedge B \longrightarrow A \vee B$
apply (*rule impI*)
apply (*erule conjE*)
apply (*rule disjI1*)
apply *assumption*
done

Detaljno objašnjen dokaz teoreme:

lemma $A \wedge B \longrightarrow A \vee B$
— *Cilj je identifikovan sa:* $A \wedge B \longrightarrow A \vee B$
— *Logičku implikaciju transformišemo u meta-implikaciju pravilom impI.*
— *Cilj je sada:* $A \wedge B \Longrightarrow A \vee B$.
apply (*rule impI*)
— *U pretpostavci se nalazi konjunkcija, eliminišemo je i dobijamo dve nezavisne činjenice pravilom conjE.*
apply (*erule conjE*)
— *Cilj je sada:* $[\![A; B]\!] \Longrightarrow A \vee B$.
— *Sada se u zaključku nalazi takva disjunkcija da je svejedno koji disjunkt ćemo ostaviti, pa na primer izdvajamo prvi, pravilom disjI1.*
apply (*rule disjI1*)
— *Cilj je sada:* $[\![A; B]\!] \Longrightarrow A$.
— *Cilj se nalazi u pretpostavkama pa primenjujemo naredbu assumption.*
apply *assumption*
— *Kraj dokaza.*
done
— *Teorema se evidentira u Isabelle/HOL sistemu.*

Zadatak 3.4. $(A \wedge B \longrightarrow C) \longrightarrow (A \longrightarrow (B \longrightarrow C))$

Dokaz teoreme:

lemma $(A \wedge B \longrightarrow C) \longrightarrow (A \longrightarrow (B \longrightarrow C))$
apply (*rule impI*)
apply (*rule impI*)
apply (*rule impI*)
apply (*erule impE*)
apply (*rule conjI*)
apply *assumption*
apply *assumption*
apply *assumption*
done

Detaljno objašnjen dokaz teoreme:

lemma $(A \wedge B \longrightarrow C) \longrightarrow (A \longrightarrow (B \longrightarrow C))$
— *Cilj je identifikovan sa:* $(A \wedge B \longrightarrow C) \longrightarrow (A \longrightarrow (B \longrightarrow C))$
— *Pošto sada imamo u zaključku nekoliko vezanih logičkih implikacija, transformišemo ih u meta-implikacije uzastopnom primenom pravila impI. Svaka primena pravila impI prebacuje pretpostavku sa desne strane* $\Longrightarrow$ *simbola na levu stranu.*
apply (*rule impI*)
— *Cilj je sada:* $A \wedge B \longrightarrow C \Longrightarrow A \longrightarrow B \longrightarrow C$.

apply (*rule impI*)
— *Cilj je sada:* $[\![A \wedge B \longrightarrow C;\ A]\!] \Longrightarrow B \longrightarrow C$.
apply (*rule impI*)
— *Cilj je sada:* $[\![A \wedge B \longrightarrow C;\ A;\ B]\!] \Longrightarrow C$.
— *Sada je potrebno eliminisati implikaciju iz pretpostavki.*
— *Da bi se eliminisala implikacija, treba prvo pokazati da možemo da dokažemo da važi njena pretpostavka (u ovom slučaju* $A \wedge B$*) na osnovu preostalih pretpostavki; pa onda dokazati da iz preostalih pretpostavki (igrom slučaja isto* $A \wedge B$*) i njenog cilja (*C*) važi globalni cilj (igrom slučaja isto* C*). Tako da nakon primene pravila impE dobijamo dva podcilja.*
apply (*erule impE*)
— *Sada dobijamo dva cilja:.*
— *1.* $[\![A;\ B]\!] \Longrightarrow A \wedge B$.
— *2.* $[\![A;\ B;\ C]\!] \Longrightarrow C$.
— *Prvo dokazujemo prvi cilj:* $[\![A;\ B]\!] \Longrightarrow A \wedge B$ *i primenjujemo pravilo uvođenja konjunkcije conjI.*
apply (*rule conjI*)
— *Sada se prvi cilj zamenjuje sa dva nova cilja:*
— *1.* $[\![A;\ B]\!] \Longrightarrow A$.
— *2.* $[\![A;\ B]\!] \Longrightarrow B$.
— *Pošto se oba cilja već nalaze u pretpostavkama, koristimo dva puta naredbu assumption.*
apply *assumption*
apply *assumption*
— *Ovim je rešen prvi cilj, ostaje drugi cilj:* $[\![A;\ B;\ C]\!] \Longrightarrow C$ *koji takođe rešavamo naredbom assumption.*
apply *assumption*
— *Kraj dokaza.*
done

Zadatak 3.5. $(A \longrightarrow (B \longrightarrow C)) \longrightarrow (A \wedge B \longrightarrow C)$

Prikazaćemo dva načina na koji možemo da dokažemo ovu lemu. Naime, u situaciji kada imamo više pretpostavki, korisnik može da bira koju pretpostavku će prvo eliminisati (odnosno koji veznik će prvo eliminisati). Naredna dva dokaza se razlikuju počevši od trećeg koraka. Dokazi su slični ali drugi dokaz je za nijansu kraći (zato što se u njemu samo jednom radi eliminacija konjunkcije).

Prvi način:

Dokaz teoreme:

lemma $(A \longrightarrow (B \longrightarrow C)) \longrightarrow (A \wedge B \longrightarrow C)$
apply (*rule impI*)

apply (*rule impI*)
apply (*erule impE*)
apply (*erule conjE*)
apply *assumption*
apply (*erule impE*)
apply (*erule conjE*)
apply *assumption*
apply *assumption*
done

Detaljno objašnjen dokaz teoreme:

Isabelle/HOL ne ispisuje zagrade koje se javljaju u izrazima ako taj raspored zagrada odgovara prioritetu operatora. Tako da u narednom detaljno raspisanom dokazu neće postojati zagrade u zaključku tvrđenja kao deo Isabelle/HOL interpretacije.

lemma $(A \longrightarrow (B \longrightarrow C)) \longrightarrow (A \wedge B \longrightarrow C)$
— *Cilj je identifikovan sa:* $(A \longrightarrow B \longrightarrow C) \longrightarrow A \wedge B \longrightarrow C$. *Sada treba dva puta implikaciju da transformišemo u meta-implikaciju primenom pravila impI.*
apply (*rule impI*)
— *Cilj je sada:* $A \longrightarrow B \longrightarrow C \Longrightarrow A \wedge B \longrightarrow C$.
apply (*rule impI*)
— *Cilj je sada:* $[\![A \longrightarrow B \longrightarrow C;\ A \wedge B]\!] \Longrightarrow C$.
— *U prvoj verziji dokaza ove leme biramo da se prvo oslobodimo implikacije u prvoj pretpostavci pravilom impE. Prvo treba da pokažemo da njena pretpostavka (A) može da se dokaže iz preostalih pretpostavki (A ∧ B), pa nakon toga treba da pokažemo da iz njenog zaključka (B ⟶ C) i preostalih pretpostavki može da se izvede finalni cilj.*
— Odnosno: leva strana implikacije postaje zaključak prvog podcilja, desna strana implikacije postaje pretpostavka drugog podcilja.
apply (*erule impE*)
— *Sada dobijamo dva cilja:.*
— *1.* $A \wedge B \Longrightarrow A$.
— *2.* $[\![A \wedge B;\ B \longrightarrow C]\!] \Longrightarrow C$.
— *Sada prvo dokazujemo prvi podcilj:* $A \wedge B \Longrightarrow A$ *i eliminišemo konjunkciju i izdvajamo konjunkte da bismo mogli da ih koristimo nezavisno.*
apply (*erule conjE*)
— *Kako se cilj sada nalazi u pretpostavkama, primenjujemo pravilo assumption.*
apply *assumption*
— *Sada dokazujemo drugi podcilj:* $[\![A \wedge B;\ B \longrightarrow C]\!] \Longrightarrow C$. *Prvo ćemo eliminisati implikaciju iz druge pretpostavke, što znači da treba iz preostale pretpostavke (A ∧ B) da dokažemo pretpostavku implikacije (B), pa zatim da iz preostale pretpostavke i zaključka implikacije (C) dokažemo konačni cilj (C).*
apply (*erule impE*)

— *Sada dobijamo dva cilja:.*
— *1.* $A \wedge B \Longrightarrow B$.
— *2.* $[\![A \wedge B;\ C]\!] \Longrightarrow C$.
— *Prvo dokazujemo prvi cilj:* $A \wedge B \Longrightarrow B$ *i primenjujemo eliminaciju konjunkcije conjE.*
apply (*erule conjE*)
— *Sada se tekući zaključak nalazi u pretpostavkama i primenjujemo pravilo assumption.*
apply *assumption*
— *Sada dokazujemo drugi cilj:* $[\![A \wedge B;\ C]\!] \Longrightarrow C$ *i vidimo da se zaključak već nalazi u pretpostavkama i ponovo primenjujemo assumption.*
apply *assumption*
— *Kraj dokaza.*
done

Drugi način:

lemma $(A \longrightarrow (B \longrightarrow C)) \longrightarrow (A \wedge B \longrightarrow C)$
apply (*rule impI*)
apply (*rule impI*)
apply (*erule conjE*) — prvo možemo da eliminišemo konjunkciju
apply (*erule impE*)
apply *assumption*
apply (*erule impE*)
apply *assumption*
apply *assumption*
done

Detaljno objašnjen dokaz teoreme:

lemma $(A \longrightarrow (B \longrightarrow C)) \longrightarrow (A \wedge B \longrightarrow C)$
— *Cilj je identifikovan sa:* $(A \longrightarrow B \longrightarrow C) \longrightarrow A \wedge B \longrightarrow C$. *Sada treba dva puta implikaciju da transformišemo u meta-implikaciju primenom pravila impI.*
apply (*rule impI*)
— *Cilj je sada:* $A \longrightarrow B \longrightarrow C \Longrightarrow A \wedge B \longrightarrow C$.
apply (*rule impI*)
— *Cilj je sada:* $[\![A \longrightarrow B \longrightarrow C;\ A \wedge B]\!] \Longrightarrow C$.
— *U drugoj verziji dokaza ove leme biramo da se prvo oslobodimo konjunkcije u drugoj pretpostavci.*
apply (*erule conjE*)
— *Sada se oslobađamo implikacije u prvoj pretpostavci pravilom impE. Prvo treba da pokažemo da njena pretpostavka (*A*) može da se dokaže iz preostalih pretpostavki (*$[\![A;\ B]\!]$*), pa nakon toga treba da pokažemo da iz njenog zaključka (*$B \longrightarrow C$*) i preostalih pretpostavki može da se izvede finalni cilj.*
apply (*erule impE*)

— *Sada dobijamo dva cilja:.*
— *1.* $[\![A;\ B]\!] \Longrightarrow A$.
— *2.* $[\![A;\ B;\ B \longrightarrow C]\!] \Longrightarrow C$.
— *Sada prvo dokazujemo prvi podcilj:* $[\![A;\ B]\!] \Longrightarrow A$. *Kako se cilj sada nalazi u pretpostavkama, primenjujemo pravilo assumption.*
apply *assumption*
— *Sada dokazujemo drugi cilj:* $[\![A;\ B;\ B \longrightarrow C]\!] \Longrightarrow C$.
— *Elimišemo implikaciju iz treće pretpostavke pravilom impE. Prvo pokazujemo da njena pretpostavka (B) može da se dokaže iz preostalih pretpostavki (*$[\![A;\ B]\!]$*), pa nakon toga da iz preostalih pretpostavki i njenog zaključka (C) može da se dokaže konačni cilj (C).*
apply (*erule impE*)
— *Sada dobijamo dva cilja:.*
— *1.* $[\![A;\ B]\!] \Longrightarrow B$.
— *2.* $[\![A;\ B;\ C]\!] \Longrightarrow C$.
— *Sada prvo dokazujemo prvi podcilj:* $[\![A;\ B]\!] \Longrightarrow B$. *Kako se cilj sada nalazi u pretpostavkama, primenjujemo pravilo assumption.*
apply *assumption*
— *Sada dokazujemo drugi cilj:* $[\![A;\ B;\ C]\!] \Longrightarrow C$ *i vidimo da se zaključak već nalazi u pretpostavkama i ponovo primenjujemo assumption.*
apply *assumption*
— *Kraj dokaza.*
done

Zadatak 3.6. $\neg\ (A \vee B) \longrightarrow \neg\ A \wedge \neg\ B$

Dokaz teoreme:

lemma shows $\neg\ (A \vee B) \longrightarrow \neg\ A \wedge \neg\ B$
apply (*rule impI*)
apply (*rule conjI*)
apply (*rule notI*)
apply (*erule notE*)
apply (*rule disjI1*)
apply *assumption*
apply (*rule notI*)
apply (*erule notE*)
apply (*rule disjI2*)
apply *assumption*
done

Detaljno objašnjen dokaz teoreme:

lemma shows $\neg\ (A \vee B) \longrightarrow \neg\ A \wedge \neg\ B$

— *Cilj je identifikovan sa:* $\neg\ (A \vee B) \longrightarrow \neg\ A \wedge \neg\ B$. *Prvo implikaciju transformišemo u meta-implikaciju primenom pravila impI.*
apply (*rule impI*)
— *Cilj je sada:* $\neg\ (A \vee B) \Longrightarrow \neg\ A \wedge \neg\ B$. *Sada primenjujemo pravilo uvođenja konjunkcije conjI.*
apply (*rule conjI*)
— *Sada dobijamo dva cilja:.*
— *1.* $\neg\ (A \vee B) \Longrightarrow \neg\ A$.
— *2.* $\neg\ (A \vee B) \Longrightarrow \neg\ B$.
— *I prvo dokazujemo prvi cilj:* $\neg\ (A \vee B) \Longrightarrow \neg\ A$. *Sada u zaključku imamo negaciju pa primenjujemo pravilo notI koje izgleda ovako:* $(P \Longrightarrow False) \Longrightarrow \neg\ P$, *znači zaključak (bez negacije) prebacujemo u pretpostavke i pokušavamo da izvedemo kontradikciju (odnosno False).*
apply (*rule notI*)
— *Cilj je sada:* $[\![\neg\ (A \vee B);\ A]\!] \Longrightarrow False$.
— *Sada iz prve pretpostavke treba da eliminišemo negaciju i koristimo pravilo notE koje izgleda ovako:* $[\![\neg\ P;\ P]\!] \Longrightarrow R$.
— Da bismo njega mogli da primenimo, potrebno je da dokažemo pozitivan oblik te pretpostavke iz preostalih pretpostavke
apply (*erule notE*)
— *Dobijamo sledeći cilj:* $A \Longrightarrow A \vee B$. *Kako je prvi disjunkt već prisutan u pretpostavkama, primenjujemo pravilo disjI1 koje će izdvojiti upravo prvi disjunkt.*
apply (*rule disjI1*)
— *Sada se zaključak nalazi u pretpostavkama i primenjujemo pravilo assumption.*
apply *assumption*
— *Sada dokazujemo drugi cilj:* $\neg\ (A \vee B) \Longrightarrow \neg\ B$. *U zaključku ponovo imamo negaciju pa primenjujemo pravilo notI:* $(P \Longrightarrow False) \Longrightarrow \neg\ P$, *i zaključak (bez negacije) prebacujemo u pretpostavke i pokušavamo da izvedemo kontradikciju (odnosno False) na osnovu njega i ostalih pretpostavki.*
apply (*rule notI*)
— *Sada dobijamo cilj:* $[\![\neg\ (A \vee B);\ B]\!] \Longrightarrow False$.
— *Sada iz prve pretpostavke treba da eliminišemo negaciju i koristimo pravilo notE:* $[\![\neg\ P;\ P]\!] \Longrightarrow R$. *I pokušavamo da iz ostalih pretpostavki dobijemo pozitivan oblik.*
apply (*erule notE*)
— Sada dobijamo cilj: $B \Longrightarrow A \vee B$. Kako je drugi disjunkt već prisutan u pretpostavkama, primenjujemo pravilo *disjI2* koje će izdvojiti upravo prvi disjunkt.
apply (*rule disjI2*)
— *Sada se zaključak nalazi u pretpostavkama i primenjujemo pravilo assumption.*
apply *assumption*
— *Kraj dokaza.*
done

Zadatak 3.7. $\neg\ A \wedge \neg\ B \longrightarrow \neg\ (A \vee B)$

I ovu teoremu ćemo dokazati na dva načina. U prvom dokazu ćemo prikazati dokaz dobijen direktno primenom pravila, a u drugom dokazu ćemo prikazati kako možemo da koristimo naredbu *back* da usmerimo primenu pravila na konkretnu pretpostavku.

Prvi način:

Dokaz teoreme:

lemma shows $\neg\ A \wedge \neg\ B \longrightarrow \neg\ (A \vee B)$
apply (*rule impI*)
apply (*rule notI*)
apply (*erule conjE*)
apply (*erule disjE*)
apply (*erule notE*)
apply *assumption*
apply (*erule notE*)
apply (*erule notE*)
apply *assumption*
done

Detaljno objašnjen dokaz teoreme:

lemma shows $\neg\ A \wedge \neg\ B \longrightarrow \neg\ (A \vee B)$
— *Cilj je identifikovan sa:* $\neg\ A \wedge \neg\ B \longrightarrow \neg\ (A \vee B)$. *Prvo implikaciju transformišemo u meta-implikaciju primenom pravila impI.*
apply (*rule impI*)
— *Cilj je sada:* $\neg\ A \wedge \neg\ B \Longrightarrow \neg\ (A \vee B)$.
— *U zaključku imamo negaciju pa primenjujemo pravilo notI i zaključak (bez negacije) prebacujemo u pretpostavke i pokušavamo da izvedemo kontradikciju (odnosno False).*
apply (*rule notI*)
— *Cilj je sada:* $[\![\neg\ A \wedge \neg\ B;\ A \vee B]\!] \Longrightarrow$ *False.*
— *Sada prvo eliminišemo konjunkciju sa leve strane, da bi mogli da pridjemo pojedinačnim konjunktima* $\neg\ A$ *i* $\neg\ B$.
apply (*erule conjE*)
— *Cilj je sada:* $[\![A \vee B;\ \neg\ A;\ \neg\ B]\!] \Longrightarrow$ *False.*
— *Sada eleminišemo disjunkciju u pretpostavkama pravilom disjE.*
apply (*erule disjE*)
— *Sada dobijamo dva cilja:.*
— *1.* $[\![\neg\ A;\ \neg\ B;\ A]\!] \Longrightarrow$ *False.*
— *2.* $[\![\neg\ A;\ \neg\ B;\ B]\!] \Longrightarrow$ *False.*
— *Sada prvo dokazujemo prvi cilj:* $[\![\neg\ A;\ \neg\ B;\ A]\!] \Longrightarrow$ *False.*
— *Pokušavamo da eliminišemo negaciju iz prve pretpostavke pravilom notE.*
apply (*erule notE*)
— *Sada je prvi cilj:* $[\![\neg\ B;\ A]\!] \Longrightarrow A$.

— Primetimo da je pravilo notE primenjeno na pretpostavku $\neg\ A$ *i da je pretpostavka* $\neg\ B$ *ostala za kasnije.*
— Kako je sada tekući zaključak već u pretpostavkama, primenjujemo pravilo assumption.
apply *assumption*
— Sada dokazujemo drugi cilj: $[\![\neg\ A;\ \neg\ B;\ B]\!] \Longrightarrow False$.
— Ponovo primenjujemo pravilo notE koje se opet primenjuje prvo na pretpostavku $\neg\ A$.
apply (*erule notE*)
— Sada je cilj: $[\![\neg\ B;\ B]\!] \Longrightarrow A$. *Pa ponovo primenjujemo pravilo notE, ali sada na pretpostavku* $\neg\ B$.
apply (*erule notE*)
— Sada je cilj: $B \Longrightarrow B$ *i možemo primeniti pravilo assumption.*
apply *assumption*
— Kraj dokaza.
done

Drugi način:

lemma shows $\neg\ A \wedge \neg\ B \longrightarrow \neg\ (A \vee B)$
apply (*rule impI*)
apply (*rule notI*)
apply (*erule conjE*)
apply (*erule disjE*)
apply (*erule notE*)
apply *assumption*
apply (*erule notE*)
back
apply *assumption*
done

Detaljno objašnjen dokaz teoreme, tek od dela koji se razlikuje od prethodnog dokaza:

lemma shows $\neg\ A \wedge \neg\ B \longrightarrow \neg\ (A \vee B)$
apply (*rule impI*)
apply (*rule notI*)
apply (*erule conjE*)
apply (*erule disjE*)
apply (*erule notE*)
apply *assumption*
— U trenutku kada je cilj: $[\![\neg\ A;\ \neg\ B;\ B]\!] \Longrightarrow False$, *svakako moramo primeniti pravilo notE. Ono se primenjuje na* $\neg\ A$ *i prebacuje* A *na desnu stranu implikacije.*
apply (*erule notE*)
— Dobijamo cilj: $[\![\neg\ B;\ B]\!] \Longrightarrow A$.

— *Međutim to nije najkraći put i možemo da iskoristimo naredbu back.*
— *Naredba back izvršava backtracking nad rezultatom prethodne naredbe u dokazu. Ova komanda će pokušati izvršenje iste naredbe ali nad sledećom pretpostavkom koja se nalazi u pretpostavkama tekućeg cilja.*
back
— *Sada dobijamo cilj:* $[\![\neg\ A;\ B]\!] \Longrightarrow B$. *I možemo odmah da primenimo naredbu assumption.*
apply *assumption*
— *Kraj dokaza.*
done

Zadatak 3.8. $\neg\ (A \longleftrightarrow \neg\ A)$

Dokaz teoreme:

lemma $\neg\ (A \longleftrightarrow \neg\ A)$
apply (*rule notI*)
apply (*erule iffE*)
apply (*erule impE*)
— apply (erule impE) — ne uspeva
back
— apply (erule impE) — ne uspeva
apply (*rule notI*)
apply (*erule impE*)
apply *assumption*
apply (*erule notE*)
apply *assumption*
apply (*erule impE*)
apply *assumption*
apply (*erule notE*)
apply *assumption*
done

Detaljno objašnjen dokaz teoreme:

U ovom dokazu se koristi pravilo za eliminaciju ekvivalencije *iffE*.

lemma $\neg\ (A \longleftrightarrow \neg\ A)$
— *Cilj je identifikovan sa:* $A \neq (\neg\ A)$.
— *Sada imamo situaciju kada je dominantni veznik negacija pa prvo primenjujemo pravilo notI.*
apply (*rule notI*)
— *Sada je cilj:* $A = (\neg\ A) \Longrightarrow$ *False.*
— *U pretpostavci imamo jednakost, odnosno ekvivalenciju pa moramo da primenimo pravilo iffE za eliminaciju ekvivalencije.*

apply (*erule iffE*)
— *Sada dobijamo cilj:* $[\![A \longrightarrow \neg\ A;\ \neg\ A \longrightarrow A]\!] \Longrightarrow False.$
— *U ovoj situaciji imamo dve implikacije pa imamo mogućnost da se oslobodimo prve ili druge implikacije. U svakom slučaju primenjujemo pravilo impE.*
apply (*erule impE*)
— *Kada prvi put primenimo pravilo impE eliminiše se prva implikacija* $A \longrightarrow \neg\ A$*; i dobijaju se naredna dva podcilja: (1) prvo dokaži njenu pretpostavku (A) iz preostalih pretpostavki; (2) nakon toga iz preostalih pretpostavki zajedno sa njenim zaključkom (*$\neg\ A$*) dokaži preostali cilj (False)*

— *Sada dobijamo dva cilja:.*
— *1.* $\neg\ A \longrightarrow A \Longrightarrow A.$
— *2.* $[\![\neg\ A \longrightarrow A;\ \neg\ A]\!] \Longrightarrow False.$

— *apply* (*erule impE*) — ne uspeva
— *Ako bismo ovako uradili, naredni korak bi morao da bude apply* (*erule impE*) *ali onda dobijamo cilj koji ne možemo da dokažemo (*$\neg\ A$*) i moramo da se vratimo! i pozivamo naredbu back.*
back

— *Naredba back poništava tu eliminaciju i eliminiše drugu implikaciju* $\neg\ A \longrightarrow A$*; i dobijamo naredna dva podcilja: (1) prvo dokaži njenu pretpostavku* $\neg\ A$ *iz preostalih pretpostavki; (2) nakon toga iz preostalih pretpostavki zajedno sa njenim zaključkom (*A*) dokaži cilj*

— *Sada dobijamo dva cilja:.*
— *1.* $A \longrightarrow \neg\ A \Longrightarrow \neg\ A.$
— *2.* $[\![A \longrightarrow \neg\ A;\ A]\!] \Longrightarrow False.$

— *Sada imamo implikaciju u pretpostavci i negaciju u zaključku. Ako bismo primenili pravilo impE za eliminaciju iz pretpostavke opet bismo dobili cilj koji ne možemo da dokažemo.*
— *apply* (*erule impE*) — ne uspeva
— *Pa primenjujemo pravilo notI.*
apply (*rule notI*)
— *Sada dobijamo cilj:* $[\![A \longrightarrow \neg\ A;\ A]\!] \Longrightarrow False$*, pa nakon toga ponovo eliminišemo implikaciju u pretpostavkama.*
apply (*erule impE*)
— *Sada dobijamo dva nova cilja:.*
— *1.* $A \Longrightarrow A.$
— *2.* $[\![A;\ \neg\ A]\!] \Longrightarrow False.$
— *Prvi cilj je trivijalan.*
apply *assumption*

— *A za drugi cilj primenjujemo pravilo notE.*
apply (*erule notE*)
— *Čime dobijamo ponovo trivijalan cilj:* $A \Longrightarrow A$.
apply *assumption*
— *Sada dokazujemo drugi glavni podcilj:* $[\![A \longrightarrow \neg\ A;\ A]\!] \Longrightarrow$ *False i prvo eliminišemo implikaciju iz pretpostavke.*
apply (*erule impE*)
— *Sada dobijamo dva nova cilja (ista kao malopre, pa se dokaz izvršava na isti način):.*
— *1.* $A \Longrightarrow A$.
— *2.* $[\![A;\ \neg\ A]\!] \Longrightarrow$ *False.*
apply *assumption*
apply (*erule notE*)
apply *assumption*
— *Kraj dokaza.*
done

3.4.1 Dodatni primeri

Naredne teoreme dokazati uz pomoć pravila prirodne dedukcije. Samo u prva dva dokaza će u komentarima biti ispisano stanje nakon svakog koraka, zbog lakšeg praćenja, ostali dokazi su navedeni bez dodatnih objašnjenja.

Zadatak 3.9. $(Q \longrightarrow R) \wedge (R \longrightarrow P \wedge Q) \wedge (P \longrightarrow Q \vee R) \longrightarrow (P \longleftrightarrow Q)$

lemma $(Q \longrightarrow R) \wedge (R \longrightarrow P \wedge Q) \wedge (P \longrightarrow Q \vee R) \longrightarrow (P \longleftrightarrow Q)$
apply (*rule impI*)
— *1.* $(Q \longrightarrow R) \wedge (R \longrightarrow P \wedge Q) \wedge (P \longrightarrow Q \vee R) \Longrightarrow P = Q$
apply (*erule conjE*)
— *1.* $[\![Q \longrightarrow R;\ (R \longrightarrow P \wedge Q) \wedge (P \longrightarrow Q \vee R)]\!] \Longrightarrow P = Q$
apply (*erule conjE*)
— *1.* $[\![Q \longrightarrow R;\ R \longrightarrow P \wedge Q;\ P \longrightarrow Q \vee R]\!] \Longrightarrow P = Q$
— *Sada primenjujemo pravilo uvođenja ekvivalencije i dobijamo naredna dva cilja: prvo iz prethodnih pretpostavki i pretpostavke* P *dokazujemo da važi* Q*, pa nakon toga da iz prethodnih pretpostavki i pretpostavke* Q *dokazujemo da važi* P.
apply (*rule iffI*)
— *1.* $[\![Q \longrightarrow R;\ R \longrightarrow P \wedge Q;\ P \longrightarrow Q \vee R;\ P]\!] \Longrightarrow Q$
— *2.* $[\![Q \longrightarrow R;\ R \longrightarrow P \wedge Q;\ P \longrightarrow Q \vee R;\ Q]\!] \Longrightarrow P$
— *Sada dokazujemo prvi cilj.*
— *U pretpostavkama prvog cilja imamo* P *i* $P \longrightarrow Q \vee R$ *pa nam odgovara da eliminišemo baš tu implikaciju. Pošto je to treća implikacija u pretpostavkama, moraćemo dva puta da pozovemo naredbu back.*

apply (*erule impE*)
— *1.* $[\![R \longrightarrow P \wedge Q;\ P \longrightarrow Q \vee R;\ P]\!] \Longrightarrow Q$
— *2.* $[\![R \longrightarrow P \wedge Q;\ P \longrightarrow Q \vee R;\ P;\ R]\!] \Longrightarrow Q$
— *3.* $[\![Q \longrightarrow R;\ R \longrightarrow P \wedge Q;\ P \longrightarrow Q \vee R;\ Q]\!] \Longrightarrow P$
back
— *1.* $[\![Q \longrightarrow R;\ P \longrightarrow Q \vee R;\ P]\!] \Longrightarrow R$
— *2.* $[\![Q \longrightarrow R;\ P \longrightarrow Q \vee R;\ P;\ P \wedge Q]\!] \Longrightarrow Q$
— *3.* $[\![Q \longrightarrow R;\ R \longrightarrow P \wedge Q;\ P \longrightarrow Q \vee R;\ Q]\!] \Longrightarrow P$
back
— *Sada smo došli do željenog rezultata.*
— *1.* $[\![Q \longrightarrow R;\ R \longrightarrow P \wedge Q;\ P]\!] \Longrightarrow P$
— *2.* $[\![Q \longrightarrow R;\ R \longrightarrow P \wedge Q;\ P;\ Q \vee R]\!] \Longrightarrow Q$
— *3.* $[\![Q \longrightarrow R;\ R \longrightarrow P \wedge Q;\ P \longrightarrow Q \vee R;\ Q]\!] \Longrightarrow P$
apply *assumption*
— *1.* $[\![Q \longrightarrow R;\ R \longrightarrow P \wedge Q;\ P;\ Q \vee R]\!] \Longrightarrow Q$
— *2.* $[\![Q \longrightarrow R;\ R \longrightarrow P \wedge Q;\ P \longrightarrow Q \vee R;\ Q]\!] \Longrightarrow P$
apply (*erule disjE*)
— *1.* $[\![Q \longrightarrow R;\ R \longrightarrow P \wedge Q;\ P;\ Q]\!] \Longrightarrow Q$
— *2.* $[\![Q \longrightarrow R;\ R \longrightarrow P \wedge Q;\ P;\ R]\!] \Longrightarrow Q$
— *3.* $[\![Q \longrightarrow R;\ R \longrightarrow P \wedge Q;\ P \longrightarrow Q \vee R;\ Q]\!] \Longrightarrow P$
apply *assumption*
— *1.* $[\![Q \longrightarrow R;\ R \longrightarrow P \wedge Q;\ P;\ R]\!] \Longrightarrow Q$
— *2.* $[\![Q \longrightarrow R;\ R \longrightarrow P \wedge Q;\ P \longrightarrow Q \vee R;\ Q]\!] \Longrightarrow P$
— *U pretpostavkama imamo P, R i dve implikacije, odgovara nam da eliminišemo onu implikaciju čije pretpostavke već znamo, odnosno drugu implikaciju, pa ćemo jednom primeniti naredbu back.*
apply (*erule impE*)
— *1.* $[\![R \longrightarrow P \wedge Q;\ P;\ R]\!] \Longrightarrow Q$
— *2.* $[\![R \longrightarrow P \wedge Q;\ P;\ R;\ R]\!] \Longrightarrow Q$
— *3.* $[\![Q \longrightarrow R;\ R \longrightarrow P \wedge Q;\ P \longrightarrow Q \vee R;\ Q]\!] \Longrightarrow P$
back
— *1.* $[\![Q \longrightarrow R;\ P;\ R]\!] \Longrightarrow R$
— *2.* $[\![Q \longrightarrow R;\ P;\ R;\ P \wedge Q]\!] \Longrightarrow Q$
— *3.* $[\![Q \longrightarrow R;\ R \longrightarrow P \wedge Q;\ P \longrightarrow Q \vee R;\ Q]\!] \Longrightarrow P$
apply *assumption*
— *1.* $[\![Q \longrightarrow R;\ P;\ R;\ P \wedge Q]\!] \Longrightarrow Q$
— *2.* $[\![Q \longrightarrow R;\ R \longrightarrow P \wedge Q;\ P \longrightarrow Q \vee R;\ Q]\!] \Longrightarrow P$
apply (*erule conjE*)
— *1.* $[\![Q \longrightarrow R;\ P;\ R;\ P;\ Q]\!] \Longrightarrow Q$
— *2.* $[\![Q \longrightarrow R;\ R \longrightarrow P \wedge Q;\ P \longrightarrow Q \vee R;\ Q]\!] \Longrightarrow P$
apply *assumption*
— *Tek sada smo dokazali da važi Q, tj. prvi podcilj*
— *Sada dokazujemo drugi podcilj*

— *1.* $[\![Q \longrightarrow R;\ R \longrightarrow P \wedge Q;\ P \longrightarrow Q \vee R;\ Q]\!] \Longrightarrow P$
— *U pretpostavkama imamo Q pa nam odgovara da se eliminiše prva implikacija.*
apply (*erule impE*)
— *1.* $[\![R \longrightarrow P \wedge Q;\ P \longrightarrow Q \vee R;\ Q]\!] \Longrightarrow Q$
— *2.* $[\![R \longrightarrow P \wedge Q;\ P \longrightarrow Q \vee R;\ Q;\ R]\!] \Longrightarrow P$
apply *assumption*
— *1.* $[\![R \longrightarrow P \wedge Q;\ P \longrightarrow Q \vee R;\ Q;\ R]\!] \Longrightarrow P$
— *U pretpostavkama imamo R pa nam odgovara da se eliminiše prva implikacija.*
apply (*erule impE*)
— *1.* $[\![P \longrightarrow Q \vee R;\ Q;\ R]\!] \Longrightarrow R$
— *2.* $[\![P \longrightarrow Q \vee R;\ Q;\ R;\ P \wedge Q]\!] \Longrightarrow P$
apply *assumption*
— *1.* $[\![P \longrightarrow Q \vee R;\ Q;\ R;\ P \wedge Q]\!] \Longrightarrow P$
apply (*erule conjE*)
— *1.* $[\![P \longrightarrow Q \vee R;\ Q;\ R;\ P;\ Q]\!] \Longrightarrow P$
apply *assumption*
— No subgoals!
done

Zadatak 3.10. $(P \longrightarrow Q) \wedge (Q \longrightarrow R) \longrightarrow (P \longrightarrow Q \wedge R)$

lemma $(P \longrightarrow Q) \wedge (Q \longrightarrow R) \longrightarrow (P \longrightarrow Q \wedge R)$
apply (*rule impI*)
— *1.* $(P \longrightarrow Q) \wedge (Q \longrightarrow R) \Longrightarrow P \longrightarrow Q \wedge R$
apply (*rule impI*)
— *1.* $[\![(P \longrightarrow Q) \wedge (Q \longrightarrow R);\ P]\!] \Longrightarrow Q \wedge R$
apply (*erule conjE*)
— *1.* $[\![P;\ P \longrightarrow Q;\ Q \longrightarrow R]\!] \Longrightarrow Q \wedge R$
apply (*rule conjI*)
— *1.* $[\![P;\ P \longrightarrow Q;\ Q \longrightarrow R]\!] \Longrightarrow Q$
— *2.* $[\![P;\ P \longrightarrow Q;\ Q \longrightarrow R]\!] \Longrightarrow R$
apply (*erule impE*)
— *1.* $[\![P;\ Q \longrightarrow R]\!] \Longrightarrow P$
— *2.* $[\![P;\ Q \longrightarrow R;\ Q]\!] \Longrightarrow Q$
— *3.* $[\![P;\ P \longrightarrow Q;\ Q \longrightarrow R]\!] \Longrightarrow R$
apply *assumption*
— *1.* $[\![P;\ Q \longrightarrow R;\ Q]\!] \Longrightarrow Q$
— *2.* $[\![P;\ P \longrightarrow Q;\ Q \longrightarrow R]\!] \Longrightarrow R$
apply *assumption*
— *1.* $[\![P;\ P \longrightarrow Q;\ Q \longrightarrow R]\!] \Longrightarrow R$
apply (*erule impE*)
— *1.* $[\![P;\ Q \longrightarrow R]\!] \Longrightarrow P$
— *2.* $[\![P;\ Q \longrightarrow R;\ Q]\!] \Longrightarrow R$

apply *assumption*
— *1.* $[\![P;\ Q \longrightarrow R;\ Q]\!] \Longrightarrow R$
apply (*erule impE*)
— *1.* $[\![P;\ Q]\!] \Longrightarrow Q$
— *2.* $[\![P;\ Q;\ R]\!] \Longrightarrow R$
apply *assumption*
— *1.* $[\![P;\ Q;\ R]\!] \Longrightarrow R$
apply *assumption*
— No subgoals!
done

Zadatak 3.11. $(P \longrightarrow Q) \wedge \neg\ Q \longrightarrow \neg\ P$

lemma $(P \longrightarrow Q) \wedge \neg\ Q \longrightarrow \neg\ P$
apply (*rule impI*)
apply (*erule conjE*)
apply (*rule notI*)
apply (*erule impE*)
apply *assumption*
apply (*erule notE*)
apply *assumption*
done

Zadatak 3.12. $(P \longrightarrow (Q \longrightarrow R)) \longrightarrow (Q \longrightarrow (P \longrightarrow R))$

lemma $(P \longrightarrow (Q \longrightarrow R)) \longrightarrow (Q \longrightarrow (P \longrightarrow R))$
apply (*rule impI*)
apply (*rule impI*)
apply (*rule impI*)
apply (*erule impE*)
apply *assumption*
apply (*erule impE*)
apply *assumption*
apply *assumption*
done

Zadatak 3.13. $\neg\ (P \wedge \neg P)$

lemma $\neg\ (P \wedge \neg P)$
apply (*rule notI*)
apply (*erule conjE*)
apply (*erule notE*)
apply *assumption*
done

Zadatak 3.14. $A \wedge (B \vee C) \longrightarrow (A \wedge B) \vee (A \wedge C)$

lemma $A \wedge (B \vee C) \longrightarrow (A \wedge B) \vee (A \wedge C)$
apply (*rule impI*)
apply (*erule conjE*)
apply (*erule disjE*)
apply (*rule disjI1*)
apply (*rule conjI*)
apply *assumption*
apply *assumption*
apply (*rule disjI2*)
apply (*rule conjI*)
apply *assumption*
apply *assumption*
done

Zadatak 3.15. $\neg (A \wedge B) \longrightarrow (A \longrightarrow \neg B)$

lemma $\neg (A \wedge B) \longrightarrow (A \longrightarrow \neg B)$
apply (*rule impI*)
apply (*rule impI*)
apply (*rule notI*)
apply (*erule notE*)
apply (*rule conjI*)
apply *assumption*
apply *assumption*
done

Zadatak 3.16. $(A \longrightarrow C) \wedge (B \longrightarrow \neg C) \longrightarrow \neg (A \wedge B)$

lemma $(A \longrightarrow C) \wedge (B \longrightarrow \neg C) \longrightarrow \neg (A \wedge B)$
apply (*rule impI*)
apply (*rule notI*)
apply (*erule conjE*)
apply (*erule conjE*)
apply (*erule impE*)
apply *assumption*
apply (*erule impE*)
apply *assumption*
apply (*erule notE*)
apply *assumption*
done

Zadatak 3.17. $(A \wedge B) \longrightarrow ((A \longrightarrow C) \longrightarrow \neg\ (B \longrightarrow \neg\ C))$

lemma $(A \wedge B) \longrightarrow ((A \longrightarrow C) \longrightarrow \neg\ (B \longrightarrow \neg\ C))$
apply (*rule impI*)
apply (*rule impI*)
apply (*rule notI*)
apply (*erule conjE*)
apply (*erule impE*)
apply *assumption*
apply (*erule impE*)
apply *assumption*
apply (*erule notE*)
apply *assumption*
done

Zadatak 3.18. $(A \longleftrightarrow B) \longrightarrow (\neg\ A \longleftrightarrow \neg\ B)$

lemma $(A \longleftrightarrow B) \longrightarrow (\neg\ A \longleftrightarrow \neg\ B)$
apply (*rule impI*)
apply (*rule iffI*)
apply (*rule notI*)
apply (*erule notE*)
apply (*erule iffE*)
apply (*erule impE*)
apply (*erule impE*)
apply *assumption*
apply *assumption*
apply (*erule impE*)
apply *assumption*
apply *assumption*
apply (*rule notI*)
apply (*erule notE*)
apply (*erule iffE*)
apply (*erule impE*)
apply *assumption*
apply *assumption*
done

Malo kraći dokaz iste teoreme uz korišćenje naredbe *back*.

lemma $(A \longleftrightarrow B) \longrightarrow (\neg\ A \longleftrightarrow \neg\ B)$
apply (*rule impI*)
apply (*rule iffI*)
apply (*rule notI*)

```
 apply (erule notE)
  apply (erule iffE)
  apply (erule impE)
— Ako ovde upotrebimo back dobijamo kraći dokaz.
   back
   apply assumption
  apply assumption
 apply (rule notI)
 apply (erule notE)
 apply (erule iffE)
 apply (erule impE)
 apply assumption
 apply assumption
 done
```

Zadatak 3.19. $A \longrightarrow \neg \neg A$

```
lemma A ⟶ ¬ ¬ A
 apply (rule impI)
 apply (rule notI)
 apply (erule notE)
 apply assumption
 done
```

Zadatak 3.20. $\neg (A \longleftrightarrow \neg A)$

```
lemma ¬ (A ⟷ ¬ A)
 apply (rule notI)
 apply (erule iffE)
 apply (erule impE)
  back
  apply (rule notI)
  apply (erule impE)
   apply assumption
  apply (erule notE)
  apply assumption
 apply (erule impE)
  apply assumption
 apply (erule notE)
 apply assumption
 done
```

Zadatak 3.21. $(A \longrightarrow B) \longrightarrow (\neg B \longrightarrow \neg A)$

lemma $(A \longrightarrow B) \longrightarrow (\neg\ B \longrightarrow \neg\ A)$
apply (*rule impI*)
apply (*rule impI*)
apply (*rule notI*)
apply (*erule notE*)
apply (*erule impE*)
apply *assumption*
apply *assumption*
done

Zadatak 3.22. $\neg\ A \vee B \longrightarrow (A \longrightarrow B)$

lemma $\neg\ A \vee B \longrightarrow (A \longrightarrow B)$
apply (*rule impI*)
apply (*rule impI*)
apply (*erule disjE*)
apply (*erule notE*)
apply *assumption*
apply *assumption*
done

3.5 Opis mehanizma primene pravila

Primena pravila prirodne dedukcije, se pokreće ključnom rečju *apply*, najčešće sa metodima *rule* i *erule*. Pored njih, moguće je primenjivati i metode *drule* i *frule*. Umesto višestrukog navođenja istog pravila više puta za redom, može se navesti simbol + nakon samog pravila. U nastavku je detaljno objašnjen mehanizam primene ovih metoda u kombinaciji sa pravilima uvođenja i eliminacije.

- Metod *rule R* unifikuje Q sa tekućim podciljem, izbacuje taj podcilj i umesto njega uvodi n novih podciljeva $P_1, \ldots, P_n$. Pravilo *rule* se koristi za primenu pravila uvođenja.

- Metod *erule R* unifikuje Q sa tekućim podciljem i istovremeno unifikuje P_1 sa nekom pretpostavkom koja se trenutno koristi za dokazivanje tekućeg podcilja. P_1 se smatra glavnom pretpostavkom kada se primenjuje sa metodom *erule* i metodom *drule* (koji će biti opisan u narednom pasusu). Podcilj se izbacuje i umesto njega se uvodi $n - 1$ novih podciljeva $P_2, \ldots, P_n$ (koji među svojim pretpostavkama nemaju pretpostavku P_1 koja je korišćena prilikom primene pravila).

U slučaju da ne želimo da se data pretpostavka obriše, može se alternativno koristiti metod (*rule R*, *assumption*).

- Metod *drule R* unifikuje P_1 sa nekom od pretpostavki (nakon te unifikacije odgovarajuća pretpostavka se briše). Podcilj se zamenjuje sa $n-1$ novih podciljeva $P_2, \ldots, P_n$. n-ti podcilj će izgledati isto kao polazni podcilj, sa dodatom pretpostavkom Q. Ovaj metod se koristi za primenu destruktivnih pravila.

- Metod *frule R* je isti kao *drule R*, s tim što se pretpostavka koja je korišćena prilikom primene pravila ne briše.

- Alternativa je (za svako od ovih pravila) je da se pravilo primeni sa instanciranjem nekih od njegovih promenljivih, na primer:

  `rule_tac` $v_1 = t_1$ `and ... and` $v_k = t_k$ `in R`

- Dodatno pored instanciranja, ako dokaz ima nekoliko nedokazanih podciljeva, korisnik može izabrati redni broj podcilja na koji želi da se fokusira i da na njega bude primenjeno odabrano pravilo:

  rule-tac $[i]$ *R*

- Pravilo *assumption*: koristi se ako se negde u pretpostavkama nalazi P, tada smatramo da je taj podcilj dokazan.

- Komanda *apply rule*: koristi se kada želimo da dokazivač sam odabere pravilo prirodne dedukcije na osnovu trenutno aktuelnog cilja.

- Komanda *done*: koristi se kao poslednja naredba u dokazu. Primenjuje se kada više nema nedokazanih ciljeva i njome se evidentira nova teorema u Isabelle/HOL sistemu.

end

3.6 Logika prvog reda

theory *Cas3-vezbe*
imports *Main*

begin

Logika prvog reda omogućava kvantifikovanje promenljivih, odnosno korišćenje univerzalnog i egzistencijalnog kvantifikatora nad promenljivima koje se javljaju u formuli. Pored pravila za uvođenje i eliminaciju veznika koja su ista kao pravila navedena u prethodnom poglavlju, postoje i dodatna pravila uvođenja i eliminacije kvantifikatora i ona će biti opisana u ovom poglavlju. Iako se kvantifikatori mogu primeniti i na funkcije i relacije, tj. iako je objektna logika sistema Isabelle/HOL logika višeg reda mi ćemo se u narednom izlaganju fokusirati samo na formule logike prvog reda (i kvantifikatori će biti korišćeni samo uz promenljive).

U zavisnosti od toga da li se koristi aksiomatska shema iskučenja trećeg $A \vee \neg A$ (*tertium non datur*) ili ne, razlikujemo intuicionističku i klasičnu logiku. Dokazi u ovom poglavlju biće intuicionistički, a u narednom poglavlju ćemo prikazati dokaze u klasičnoj logici.

Za prirodnu dedukciju za logiku prvog reda (kada se uključe pravila za klasičnu logiku) može se pokazati saglasnost i potpunost tj. može se dokazati da su sve formule koje se mogu dokazati valjane formule (tačne su u svakom modelu), kao i da se svaka valjana formula može dokazati.

3.6.1 Pravila za univerzalni i egzistencijalni kvantifikator

Uvođenje univerzalnog kvantifikatora: U neformalnoj matematici, dokazi univerzalno kvantifikovanih tvrđenja se izvode tako što se dokaz izvede za neku fiksiranu proizvoljnu veličinu. Na primer, da bi se dokazalo da je svaki prirodan broj deljiv sa 6 deljiv i sa 2, razmotri se proizvoljni prirodan broj x, i za njega se pokaže da ako je deljiv sa 6, onda je deljiv i sa 2 (naravno, na ovom mestu se primeni pravilo uvođenja implikacije, pa se pretpostavi da je x deljiv sa 6 i iz te pretpostavke se dokaže da je x deljiv i sa 2). Proizvoljnost broja x je veoma važna i ona nam zabranjuje da bilo šta dodatno pretpostavimo za broj x. Na primer, posle fiksiranja proizvoljnog broja x, ne možemo odjednom da pretpostavimo da je on paran ili prost, jer bi se time narušila proizvoljnost broja x i opštost našeg dokaza.

Dakle, da bismo mogli da uvedemo univerzalni kvantifikator, potrebno je da formulu koja je pod kvantifikatorom dokažemo bez ikakvih pretpostavki

vezanih za promenljivu koja je kvantifikovana. Proizvoljnost te promenljive se eksplicitno izražava korišćenjem meta-logike sistema Isabelle. Ta meta-logika sadrži svoju verziju univerzalnog kvantifikatora ($\bigwedge$), koja naglašava da je promenljiva koju on vezuje proizvoljna ali fiksirana vrednost. Recimo i da je pored simbola $\Longrightarrow$ (koji se koristi za zapisivanje pravila izvođenja i odvajanje pretpostavki od zaključaka) i $\equiv$ (koji se koristi za definisanje konstanti), ovo je jedini preostali veznik koji pripada meta-logici (ona, dakle, koristi samo tri simbola).

Pravilo kojim se uvodi univerzalni kvantifikator se zove *allI*: $(\bigwedge x.\ ?P\ x) \Longrightarrow \forall\, x.\ ?P\ x$, i ovako se zapisuje u Isabelle-u. U udžbenicima matematičke logike to pravilo se obično zapisuje na sledeći način:

$$\frac{A[x \to y]}{(\forall x)A}$$

Primenjuje se kada se u zaključku teoreme javlja univerzalni kvantifikator. Kao rezultat primene ovog pravila HOL kvantifikator $\forall$ se zamenjuje kvantifikatorom na meta-nivou $\bigwedge$, i uvodi se proizvoljna ali fiksirana promenljiva x (to je sveža promenljiva, odnosno promenljiva koja se ne javlja ni u jednom delu formule koja se dokazuje). Nakon toga cilj postaje da se ostatak formule (bez kvantifikatora) dokaže za proizvoljno x (o kome nemamo nikakvih pretpostavki).

Eliminacija univerzalnog kvantifikatora: U neformalnoj matematici, kada u nekom dokazu među pretpostavkama ili ranije dokazanim tvrđenjima imamo neko univerzalno kvantifikovano tvrđenje oblika $\forall\ x.\ P\ x$, na osnovu toga možemo zaključiti da tvrđenje P važi za bilo koji objekat tj. za bilo koji term. Dakle, možemo slobodno zaključiti da važi $P\ t$, šta god da je term t. Primetimo, da za razliku od uvođenja univerzalnog kvantifikatora tj. dokaza univerzalnih svojstava, ovde ne postoji ograničenje za t. Na primer, ako dokazujemo da nešto važi za sve brojeve, ne možemo dokaz zasnovati na tome da dokazujemo to za neko x za koje već znamo da je paran broj, ali ako znamo da nešto važi za sve brojeve, to možemo primeniti i na broj x za koji već znamo da je paran broj.

U sistemu Isabelle, ako promenljiva A važi za svako x tada eliminisanjem univerzalnog kvantifikatora zamenjujemo vezanu promenljivu x, šematskom promenljivom $?x$ koja može biti zamenjena proizvoljnim termom t. Pravilo se zove *allE*: $[\![\forall\, x.\ ?P\ x;\ ?P\ ?x \Longrightarrow ?R]\!] \Longrightarrow ?R$, i ovako se zapisuje u Isabelle-u. U udžbenicima matematičke logike, to pravilo se obično zapisuje na sledeći način:

$$\frac{(\forall x)A}{A[x \to t]}$$

Uvođenje egzistencijalnog kvantifikatora: Kada u neformalnoj matematici treba da dokažemo da postoji neki objekat koji zadovoljava neko dato svojstvo, dovoljno je pronaći samo jednog svedoka tj. dovoljno je da među pretpostavkama pronađemo bilo kakav term t koji ima to željeno svojstvo P. Ni u ovom slučaju ne moramo nametati uslov proizvoljnosti terma t. Na primer, ako treba da pokažemo da postoji neki broj koji je deljiv sa 3 a znamo da je broj x deljiv sa 6, ne smeta nam to što za taj broj ujedno znamo i da je deljiv sa 2.

U sistemu Isabelle/HOL, ako možemo da za neki term t dokažemo da važi $P\ t$ (ili ako to imamo među trenutnim pretpostavkama), onda to znači da možemo da dokažemo da važi i $\exists\ x.\ P\ x$. Pravilo se zove *exI*: $?P\ ?x \Longrightarrow \exists\, x.\ ?P\ x$, i ovako se zapisuje u Isabelle-u. U udžbenicima matematičke logike to pravilo se obično zapisuje na sledeći način:

$$\frac{A[x \to t]}{(\exists x)A}$$

Eliminacija egzistencijalnog kvantifikatora: Kada u neformalnoj matematici znamo da postoji objekat koji ima neko svojstvo P, tada u dokazu možemo taj objekat da imenujemo tako što uvedemo neko proizvoljno fiksirano x i pretpostavimo da važi $P\ x$. Proizvoljnost objekta x nas sprečava da bilo šta dodatno pretpostavimo o objektu x osim da ima svojstvo P. Zaista, ako znamo (ili umemo da dokažemo) da postoji broj x koji ima neko svojstvo P, ne možemo dodatno da pretpostavimo da je taj broj neparan, jer tako nešto prosto ne znamo (možda su baš svi brojevi koji imaju svojstvo P parni).

U sistemu Isabelle/HOL, da bismo eliminisali egzistencijalni kvantifikator, dovoljno je da pod pretpostavkom da formula A važi za neki proizvoljan element y izvedemo formulu B. Pravilo se zove *exE*: $[\![\exists\, x.\ P\ x;\ \bigwedge x.\ P\ x \Longrightarrow Q]\!] \Longrightarrow Q$, i ovako se zapisuje u Isabelle-u. U udžbenicima matematičke logike to pravilo se obično zapisuje na sledeći način:

$$\frac{(\exists x)A \qquad \begin{array}{c}[A[x \to y]] \\ \vdots \\ B\end{array}}{B}$$

Dobijamo fiksirano ali proizvoljno x (o kome nemamo nikakvih dodatnih pretpostavki) označeno Isabelle kvantifikatorom $\bigwedge$, i novu pretpostavku (formulu iz koje smo izbacili egzistencijalni kvantifikator).

3.6.2 Dokazi u logici prvog reda

U narednim zadacima ćemo koristiti pravilo oslobađanja i uvođenja univerzalnog i egzistencijalnog kvantifikatora. Ova pravila se primenjuju uz pomoć metoda *rule-tac* (za uvođenje kvantifikatora) i *erule-tac* (za eliminisanje kvantifikatora). Te naredbe omogućavaju primenu pravila i istovremeno instanciranje nekih promenljivih koje se javljaju u tom pravilu. Opšti oblik je *rule-tac v1 = t1 and ... and vk = tk in R*, i odgovara primeni pravila *rule R*, uz instanciranje promenljivih *v1 ... vk* sa datim termovima. Obratite pažnju da se vrednosti promenljivih navode pod navodnicima (što neće biti vidljivo u samoj knjizi zbog načina na koji Isabelle prevodi dati tekst).

U terminima *bezbednih i nebezbednih pravila* o kojima smo već govorili, pravila *allI* i *exE* su bezbedna i treba forsirati njihovu primenu nad nebezbednim pravilima *allE* i *exI*. Dakle, kreiranje fiksiranih proizvoljnih parametara ($\bigwedge$ x), je bezbedno, za razliku od kreiranja novih promenljivih šematskih promenljivih (*?x*).

Leme koje se javljaju u ovom poglavlju su već viđene u prethodnom poglavlju ove knjige. Iako je moguće koristiti automatske alate za njihovu proveru, raspisivanje detaljnih dokaza nam omogućava da usvojimo tehniku prirodne dedukcije i njenu realizaciju u sistemu Isabelle/HOL.

Slično kao i ranije, pošto je apply-skript dokaze jako teško pratiti bez Isabelle sistema, biće prikazano stanje dokazivača nakon svakog koraka i po potrebi će biti iskomentarisani određeni koraci.

Zadatak 3.23. Ako znamo da je svaki čovek smrtan, i ako je Sokrat čovek, pokazati da je onda i Sokrat smrtan.

lemma ((∀ *x*. *covek x* ⟶ *smrtan x*) ∧ *covek Sokrat*) ⟶ *smrtan Sokrat*
apply (*rule impI*)
— *1.* (∀ *x*. *covek x* ⟶ *smrtan x*) ∧ *covek Sokrat* ⟹ *smrtan Sokrat*

apply (*erule conjE*)
— *1.* $[\![\forall x.\ covek\ x \longrightarrow smrtan\ x;\ covek\ Sokrat]\!] \Longrightarrow smrtan\ Sokrat$
— *Sada primenjujemo instanciranje, ako formula važi za svako x onda važi i za Sokrata. Ime promenljive koje se koristi u primeni pravila odgovara samom pravilu, slučajno je ovde isto ime i u formuli i u pravilu.*
apply (*erule-tac x = Sokrat* **in** *allE*)
— *1.* $[\![covek\ Sokrat;\ covek\ Sokrat \longrightarrow smrtan\ Sokrat]\!] \Longrightarrow smrtan\ Sokrat$
— Nakon toga eliminišemo implikaciju i jednostavno završavamo dokaz.
apply (*erule impE*)
— *1.* $covek\ Sokrat \Longrightarrow covek\ Sokrat$
— *2.* $[\![covek\ Sokrat;\ smrtan\ Sokrat]\!] \Longrightarrow smrtan\ Sokrat$
apply *assumption*
— *1.* $[\![covek\ Sokrat;\ smrtan\ Sokrat]\!] \Longrightarrow smrtan\ Sokrat$
apply *assumption*
— *No subgoals!*
done

Dokažimo valjanost narednih de Morganovih pravila za kvantifikatore.

lemma *de-Morgan-1*: $(\exists\ x.\ \neg\ P\ x) \longrightarrow (\neg\ (\forall\ x.\ P\ x))$
apply (*rule impI*)
— $\exists x.\ \neg\ P\ x \Longrightarrow \neg\ (\forall x.\ P\ x)$
apply (*rule notI*)
— $[\![\exists x.\ \neg\ P\ x;\ \forall x.\ P\ x]\!] \Longrightarrow False$
— *Pretpostavićemo da važi* ‹$\forall x.\ P\ x$› *i dokazaćemo kontradikciju sa* ‹$\exists\ x.\ \neg\ Px$›
apply (*erule exE*)
— *Pošto znamo da postoji element koji ne zadovoljava svojstvo* ‹P›, *eliminišemo egzistencijalni kvantifikator tako što imenujemo jedan takav element* ‹x›.
— $\bigwedge x.\ [\![\forall x.\ P\ x;\ \neg\ P\ x]\!] \Longrightarrow False$
apply (*erule-tac x = x* **in** *allE*)
— *Pošto znamo da* ‹P› *važi za svaki element, on mora da važi i za ovaj, upravo imenovani element* ‹x›. *Zato eliminišemo univerzalni kvantifikator i to tako što ga instanciramo za taj element* ‹x›.
— $\bigwedge x.\ [\![\neg\ P\ x;\ P\ x]\!] \Longrightarrow False$
apply (*erule notE*)
apply *assumption*
— *Pošto element* ‹x› *istovremeno zadovoljava i ne zadovoljava svojstvo* ‹P›, *dokaz se lako završava eliminacijom negacije.*
done

lemma *de-Morgan-2*: $(\forall\ x.\ \neg\ P\ x) \longrightarrow (\neg\ (\exists\ x.\ P\ x))$
apply (*rule impI*)
— $\forall x.\ \neg\ P\ x \Longrightarrow \nexists x.\ P\ x$
— *Slično kao malopre, primenjuje pravilo uvođenja negacije*

apply (*rule notI*)
— *Pošto znamo da postoji element koji zadovoljava svojstvo P, eliminišemo egzistencijalni kvantifikator tako što imenujemo jedan takav element x.*
apply (*erule exE*)
— *Pošto znamo da negacija svojstva P važi za svaki element, onda to isto mora važiti i za ovaj novi element x i eliminišemo univerzalni kvantifikator.*
apply (*erule-tac* $x = x$ **in** *allE*)
— *Element x istovremeno i zadovoljava i ne zadovoljava svojstvo P pa se dokaz završava eliminacijom negacije.*
apply (*erule notE*)
apply *assumption*
done

lemma *de-Morgan-3*: $(\neg\ (\exists\ x.\ P\ x)) \longrightarrow (\forall\ x.\ \neg\ P\ x)$
apply (*rule impI*)
— $\nexists x.\ P\ x \Longrightarrow \forall x.\ \neg\ P\ x$
— *Primenjujemo pravilo uvođenja univerzalnog kvantifikatora i uvodimo fiksiranu ali proizvoljnu promenljivu x*
apply (*rule allI*)
— *Sada primenjujemo pravilo uvođenja negacije i novi cilj nam postaje da dokažemo kontradikciju*
apply (*rule notI*)
— *Sada eliminišemo negaciju iz pretpostavki*
apply (*erule notE*)
— *I primenjujemo pravilo uvođenja egzistencijalnog kvantifikatora tako što upotrebimo ovo konkretno malopre uvedeno x*
apply (*rule-tac* $x = x$ **in** *exI*)
— *Sada su pretpostavka i zaključak identični.*
apply *assumption*
done

Zadatak 3.24. Prirodnom dedukcijom dokazati da je naredno tvrđenje teorema logike prvog reda:

lemma $(\exists\ x.\ P\ x) \wedge (\forall\ x.\ P\ x \longrightarrow Q\ x) \longrightarrow (\exists\ x.\ Q\ x)$
apply (*rule impI*)
— *1.* $(\exists x.\ P\ x) \wedge (\forall x.\ P\ x \longrightarrow Q\ x) \Longrightarrow \exists x.\ Q\ x$
apply (*erule conjE*)
— *1.* $[\![\exists x.\ P\ x;\ \forall x.\ P\ x \longrightarrow Q\ x]\!] \Longrightarrow \exists x.\ Q\ x$
— *Sada prvo eliminišemo egzistencijalni kvantifikator iz pretpostavki i primenjujemo pravilo exE.*
apply (*erule exE*)
— *1.* $\bigwedge x.\ [\![\forall x.\ P\ x \longrightarrow Q\ x;\ P\ x]\!] \Longrightarrow \exists x.\ Q\ x$

— *Dobija se fiksirano ali proizvoljno x, sa kojim možemo da instanciramo promenljivu x prilikom eliminacije univerzalnog kvantifikatora.*
apply (*erule-tac* $x = x$ **in** *allE*)
— *1.* $\bigwedge x.\ [\![P\ x;\ P\ x \longrightarrow Q\ x]\!] \Longrightarrow \exists\, x.\ Q\ x$
— *Sada eliminišemo implikaciju u pretpostavkama i dobijamo dva cilja.*
apply (*erule impE*)
— *1.* $\bigwedge x.\ P\ x \Longrightarrow P\ x$
— *2.* $\bigwedge x.\ [\![P\ x;\ Q\ x]\!] \Longrightarrow \exists\, x.\ Q\ x$
apply *assumption*
— *1.* $\bigwedge x.\ [\![P\ x;\ Q\ x]\!] \Longrightarrow \exists\, x.\ Q\ x$
— Sada se oslobađamo egzistencijalnog kvantifikatora sa desne strane, tako što pokušavamo sa ovim konkretnim *x*.
apply (*rule-tac* $x = x$ **in** *exI*)
— *1.* $\bigwedge x.\ [\![P\ x;\ Q\ x]\!] \Longrightarrow Q\ x$
apply *assumption*
— *No subgoals!*
done

Zadatak 3.25. Ako je svaki covek smrtan; i ako je svaki Grk čovek; onda važi da je svaki Grk smrtan. Ovo je malo komplikovanija teorema, može se zapisati sa jednom vezanom promenljivom x, ali čitljivije je ako uvedemo različite vezane promenljive: c, g i a.

lemma $(\forall\ c.\ covek\ c \longrightarrow smrtan\ c) \wedge (\forall\ g.\ grk\ g \longrightarrow covek\ g)$
$\longrightarrow (\forall\ a.\ grk\ a \longrightarrow smrtan\ a)$
— *Prvo koristimo impI pravilo, pa nakon toga eliminišemo konjunkciju iz pretpostavki.*
apply (*rule impI*)
— *1.* $(\forall\, c.\ covek\ c \longrightarrow smrtan\ c) \wedge (\forall\, g.\ grk\ g \longrightarrow covek\ g) \Longrightarrow \forall\, a.\ grk\ a \longrightarrow smrtan\ a$
apply (*erule conjE*)
— *1.* $[\![\forall\, c.\ covek\ c \longrightarrow smrtan\ c; \forall\, g.\ grk\ g \longrightarrow covek\ g]\!] \Longrightarrow \forall\, a.\ grk\ a \longrightarrow smrtan\ a$
— *Sada uvodimo proizvoljno ali konkretno a uvođenjem univerzalnog kvantifikatora.*

apply (*rule allI*)
— *1.* $\bigwedge a.\ [\![\forall\, c.\ covek\ c \longrightarrow smrtan\ c; \forall\, g.\ grk\ g \longrightarrow covek\ g]\!] \Longrightarrow grk\ a \longrightarrow smrtan\ a$
— *Sada uvodimo implikaciju sa desne strane tako što njenu pretpostavku prebacujemo na levu stranu. Dakle, da bismo dokazali da su svi Grci smrtni, to ćemo uraditi za za jednog Grka a o kome ne znamo ništa osim da je Grk.*
apply (*rule impI*)
— *1.* $\bigwedge a.\ [\![\forall\, c.\ covek\ c \longrightarrow smrtan\ c; \forall\, g.\ grk\ g \longrightarrow covek\ g; grk\ a]\!] \Longrightarrow smrtan\ a$

— Obe implikacije u pretpostavkama važe za sve ljude. Nama je potrebno da dokažemo nešto samo za osobu a, pa zato eliminišemo oba univerzalna kvantifikatora instancirajući ih baš sa a.
apply (*erule-tac* $x = a$ **in** *allE*)
— 1. $\bigwedge a.\ [\![\forall g.\ grk\ g \longrightarrow covek\ g;\ grk\ a;\ covek\ a \longrightarrow smrtan\ a]\!] \Longrightarrow smrtan\ a$
apply (*erule-tac* $x = a$ **in** *allE*)
— 1. $\bigwedge a.\ [\![grk\ a;\ covek\ a \longrightarrow smrtan\ a;\ grk\ a \longrightarrow covek\ a]\!] \Longrightarrow smrtan\ a$
— U ovom trenutku smo došli do iskaznog nivoa i možemo da eliminišemo implikacije i nastavimo dokaz na uobičajen način.
apply (*erule impE*)
— 1. $\bigwedge a.\ [\![grk\ a;\ grk\ a \longrightarrow covek\ a]\!] \Longrightarrow covek\ a$
— 2. $\bigwedge a.\ [\![grk\ a;\ grk\ a \longrightarrow covek\ a;\ smrtan\ a]\!] \Longrightarrow smrtan\ a$
apply (*erule impE*)
— 1. $\bigwedge a.\ grk\ a \Longrightarrow grk\ a$
— 2. $\bigwedge a.\ [\![grk\ a;\ covek\ a]\!] \Longrightarrow covek\ a$
— 3. $\bigwedge a.\ [\![grk\ a;\ grk\ a \longrightarrow covek\ a;\ smrtan\ a]\!] \Longrightarrow smrtan\ a$
apply *assumption*
— 1. $\bigwedge a.\ [\![grk\ a;\ covek\ a]\!] \Longrightarrow covek\ a$
— 2. $\bigwedge a.\ [\![grk\ a;\ grk\ a \longrightarrow covek\ a;\ smrtan\ a]\!] \Longrightarrow smrtan\ a$
apply *assumption*
— 1. $\bigwedge a.\ [\![grk\ a;\ grk\ a \longrightarrow covek\ a;\ smrtan\ a]\!] \Longrightarrow smrtan\ a$
apply *assumption*
— No subgoals!
done

3.6.3 Dodatni primeri

Zadatak 3.26. $(\forall\ a.\ P\ a \longrightarrow Q\ a) \wedge (\forall\ b.\ P\ b) \longrightarrow (\forall\ x.\ Q\ x)$

lemma $(\forall\ a.\ P\ a \longrightarrow Q\ a) \wedge (\forall\ b.\ P\ b) \longrightarrow (\forall\ x.\ Q\ x)$
apply (*rule impI*)
apply (*erule conjE*)
apply (*rule allI*)
apply (*erule-tac* $x = x$ **in** *allE*)
apply (*erule-tac* $x = x$ **in** *allE*)
apply (*erule impE*)
apply *assumption*
apply *assumption*
done

Zadatak 3.27. $(\exists\ x.\ A\ x \vee B\ x) \longrightarrow (\exists\ x.\ A\ x) \vee (\exists\ x.\ B\ x)$

lemma $(\exists\ x.\ A\ x \vee B\ x) \longrightarrow (\exists\ x.\ A\ x) \vee (\exists\ x.\ B\ x)$

apply (*rule impI*)
apply (*erule exE*)
— *Eliminišemo disjunkciju i razdvajamo na dve odvojene grane u dokazu.*
apply (*erule disjE*)
— *Sada naglašavamo da prvo pokušavamo da dokažemo prvi disjunkt u zaključku time što primenjujemo pravilo disjI1. Ovde je bitno primetiti da ne bismo mogli da dokažemo drugi disjunkt.*
apply (*rule disjI1*)
apply (*rule-tac* $x = x$ **in** *exI*)
apply *assumption*
— *Dokazali smo prvi podcilj, sada drugi podcilj pokušavamo da dokažemo kroz drugi disjunkt time što primenjujemo pravilo disjI2.*
apply (*rule disjI2*)
apply (*rule-tac* $x = x$ **in** *exI*)
apply *assumption*
done

Zadatak 3.28. $(\forall\ x.\ A\ x \longrightarrow \neg\ B\ x) \longrightarrow \neg\ (\exists\ x.\ A\ x \wedge B\ x)$

lemma $(\forall\ x.\ A\ x \longrightarrow \neg\ B\ x) \longrightarrow \neg\ (\exists\ x.\ A\ x \wedge B\ x)$
apply (*rule impI*)
apply (*rule notI*)
apply (*erule exE*)
apply (*erule conjE*)
apply (*erule-tac* $x = x$ **in** *allE*)
apply (*erule impE*)
apply *assumption*
apply (*erule notE*)
apply *assumption*
done

Narednih nekoliko primera je opisano rečenicama i formulama. Na ispitu će biti navedena samo formulacija uz pomoć rečenica, a zadatak će biti i zapis uz pomoć formule i raspisani dokaz same formule. Stil dokaza ćete najverovatnije moći sami da birate, sem ako nije eksplicitno navedeno koji tip dokaza se očekuje. Primetićete da su ovi dokazi ponekad lakši ako se koristi nestruktuiran stil dokaza nego struktuiran stil dokaza koji ćemo koristiti kasnije.

Zadatak 3.29. Formulisati i dokazati naredno tvrđenje:

Ako za svaki broj koji nije paran važi da je neparan;

i ako za svaki neparan broj važi da nije paran;

pokazati da onda za svaki broj važi da nije istovremeno i paran i neparan.

lemma ($\forall$ *x*. $\neg$ *paran x* $\longrightarrow$ *neparan x*) $\wedge$ ($\forall$ *x*. *neparan x* $\longrightarrow$ $\neg$ *paran x*)
$\longrightarrow$ ($\forall$ *x*. $\neg$ (*paran x* $\wedge$ *neparan x*))
apply (*rule impI*)
— *Sada primenjujemo pravilo uvođenja implikacije u zaključku.*
apply (*rule allI*)
— *Pa eliminišemo konjunkciju iz pretpostavki i dobijamo dve pretpostavke.*
apply (*erule conjE*)
— *Pa primenjujemo pravilo uvođenja negacije u zaključku tako što prebacujemo pozitivan oblik u pretpostavke.*
apply (*rule notI*)
— *Pa opet konjunkciju eliminišemo.*
apply (*erule conjE*)
— *Sada u pretpostavkama imamo dve univerzalne kvantifikovane formule, koje pošto važe za svako x važice i za ovo već uvedeno fiksirano ali proizvoljno x.*
apply (*erule-tac x* = *x* **in** *allE*)
apply (*erule-tac x* = *x* **in** *allE*)
— *Posmatranjem skupa pretpostavki vidimo da nam odgovara da eliminišemo drugu implikaciju (jer njenu levu stranu već imamo u pretpostavkama). Pa odmah možemo da zaključimo da ćemo pozvati naredbu back.*
apply (*erule impE*)
back
apply *assumption*
apply (*erule notE*)
apply *assumption*
done

Zadatak 3.30. Formulisati i dokazati naredno tvrđenje:
Ako za svaki broj koji nije paran važi da je neparan;
i ako za svaki neparan broj važi da nije paran;
pokazati da onda za svaki broj važi da je ili paran ili neparan.

lemma ($\forall$ *x*. $\neg$ *paran x* $\longrightarrow$ *neparan x*) $\wedge$ ($\forall$ *x*. *neparan x* $\longrightarrow$ $\neg$ *paran x*)
$\longrightarrow$ ($\forall$ *x*. *paran x* $\vee$ *neparan x*)
apply (*rule impI*)
— *Prvo eliminišemo konjunkciju i dobijamo dve pretpostavke.*
apply (*erule conjE*)
— *Pa primenjujemo pravilo uvođenja univerzalnog kvantifikatora.*
apply (*rule allI*)
— *Pa dva puta primenjujemo allE na obe pretpostavke.*
apply (*erule-tac x* = *x* **in** *allE*)
apply (*erule-tac x* = *x* **in** *allE*)
apply (*rule ccontr*)
— *Sada transformisemo stare pretpostavke.*

apply (*erule impE*)
apply (*rule notI*)
apply (*erule notE*)
apply (*rule disjI1*)
apply *assumption*
apply (*erule impE*)
apply *assumption*
apply (*erule notE*)
apply (*rule disjI2*)
apply *assumption*
done

Zadatak 3.31. Ako svaki konj ima potkovice;
i ako ne postoji čovek koji ima potkovice;
i ako znamo da postoji makar jedan čovek;
dokazati da postoji čovek koji nije konj.

lemma ($\forall$ *x. konj x* $\longrightarrow$ *ima-potkovice x*) $\wedge$ $\neg$ ($\exists$ *x. covek x* $\wedge$ *ima-potkovice x*)
$\wedge$ ($\exists$ *x. covek x*)
$\longrightarrow$ ($\exists$ *x. covek x* $\wedge$ $\neg$ *konj x*)
apply (*rule impI*)
— *Pošto imamo tri konjunkta, konjunkciju eliminišemo dva puta.*
apply (*erule conjE*)
apply (*erule conjE*)
— *Sada imamo samo jedan egzistencijalni kvantifikator u pretpostavkama, i njega prvog oslobađamo.*
apply (*erule exE*)
— *Sada taj novouvedeni x koristimo u zaključku.*
apply (*rule-tac x = x* **in** *exI*)
— *Sada primenjujemo pravilo uvođenja konjunkcije conjI.*
apply (*rule conjI*)
— *Rešavamo prvi podcilj.*
apply *assumption*
— *Sada rešavamo drugi podcilj.*
apply (*rule notI*)
— *Sada univerzalni kvantifikator u pretpostavkama instanciramo već uvedenim x.*

apply (*erule-tac x = x* **in** *allE*)
— *Sada se oslobađamo implikacije u pretpostavkama.*
apply (*erule impE*)
apply *assumption*
— *Sada se oslobađamo negacije.*
apply (*erule notE*)

— Sada je egzistencijalni kvantifikator prešao u zaključak i instanciramo ga već uvedenim objektom x.
apply (*rule-tac* $x = x$ **in** *exI*)
apply (*rule conjI*)
apply *assumption*
apply *assumption*
done

Zadatak 3.32. Ako je svaki kvadrat romb;
i ako je svaki kvadrat pravougaonik;
i ako znamo da postoji makar jedan kvadrat;
onda postoji makar jedan romb koji je istovremeno i pravougaonik.

lemma $(\forall\ x.\ kvadrat\ x \longrightarrow romb\ x) \wedge (\forall\ x.\ kvadrat\ x \longrightarrow pravougaonik\ x)$
$\wedge\ (\exists\ x.\ kvadrat\ x) \longrightarrow (\exists\ x.\ romb\ x \wedge pravougaonik\ x)$
apply (*rule impI*)
apply (*erule conjE*)
apply (*erule conjE*)
apply (*erule exE*)
— Ovim se uvodi x koji je kvadrat.
apply (*erule-tac* $x = x$ **in** *allE*)
apply (*erule-tac* $x = x$ **in** *allE*)
apply (*rule-tac* $x = x$ **in** *exI*)
apply (*rule conjI*)
apply (*erule impE*)
apply *assumption*
apply *assumption*
apply (*erule impE*)
apply *assumption*
apply (*erule impE*)
apply *assumption*
apply *assumption*
done

Zadatak 3.33. Ako je relacija R simetrična, tranzitivna i ako za svako x postoji y koje je sa njim u relaciji, onda je relacija R i refleksivna.

Prvo je potrebno uvesti definicije pojmova koji se javljaju u postavci ovog zadatka:

definition
$reflexive\ R \longleftrightarrow (\forall\ x.\ R\ x\ x)$

definition
transitive $R \longleftrightarrow (\forall\ x\ y\ z.\ R\ x\ y \wedge R\ y\ z \longrightarrow R\ x\ z)$

definition
symmetric $R \longleftrightarrow (\forall\ x\ y.\ R\ x\ y \longleftrightarrow R\ y\ x)$

lemma *symmetric* $R \wedge$ *transitive* $R \wedge (\forall\ x.\ \exists\ y.\ R\ x\ y) \longrightarrow$ *reflexive* R
unfolding *symmetric-def transitive-def reflexive-def*
— *prvo primenjujemo unfolding da bismo radili nad samim formulama*
apply (*rule impI*)
apply (*erule conjE*)
apply (*erule conjE*)
apply (*rule allI*)
— *uvodimo fiksirano ali proizvoljno x*
— *sada hoćemo da eliminišemo treću pretpostavku, da bismo došli do egzistencijalnog kvantifikatora.*
apply (*erule-tac* $x = x$ **in** *allE*)
back
back
apply (*erule exE*)
— *uvodimo fiksirano y*
— *sada skidamo univerzalni kvantifikator za već dobijeno x*
apply (*erule-tac* $x = x$ **in** *allE*)
apply (*erule-tac* $x = x$ **in** *allE*)
— *Sada treba da vodimo računa da nam je primarni kvantifikator po y, i odgovara nam da ostane y (naspram x, pokušajte šta bi se dobilo na taj način).*
apply (*erule-tac* $x = y$ **in** *allE*)
apply (*erule-tac* $x = y$ **in** *allE*)
— *Sada biramo sa čim ćemo da zamenimo z, odgovara nam x (pogledajte šta bi se desilo kada biste napisali y).*
apply (*erule-tac* $x = x$ **in** *allE*)
— *Sada radimo na uobičajeni nacin, nakon što smo sve unifikovali kako treba.*
apply (*erule impE*)
apply (*rule conjI*)
apply *assumption*
apply (*erule iffE*)
apply (*erule impE*)
apply *assumption*
apply *assumption*
apply *assumption*
done
end

3.7 Klasična logika

theory *Cas4-vezbe*
imports *Main HOL−Library.LaTeXsugar*

begin

3.7.1 Pravilo ccontr - klasična kontradikcija

Početkom 20. veka određen broj matematičara izrazio je svoje kritike klasične matematike. Naime, u klasičnoj matematici se često postojanje nekog objekta dokazuje svođenjem na kontradikciju, tako što se dokaže da nije moguće da takav objekat ne postoji, iako se traženi objekat ne konstruiše efektivno.

Razmotrimo naredni čuveni primer takvog dokaza. Dokazaćemo da postoje iracionalni brojevi a i b takvi da je a^b racionalan broj. Razmotrimo broj $\sqrt{2}^{\sqrt{2}}$. Ako je on racionalan, tada je dokaz završen, jer možemo uzeti $a = \sqrt{2}$ i $b = \sqrt{2}$, znajući da je $\sqrt{2}$ iracionalan broj. Ako on nije racionalan, tada možemo uzeti da je $a = \sqrt{2}^{\sqrt{2}}$ i $b = \sqrt{2}$. I jedan i drugi broj je iracionalan, dok je $a^b = (\sqrt{2}^{\sqrt{2}})^{\sqrt{2}} = \sqrt{2}^{(\sqrt{2}\sqrt{2})} = \sqrt{2}^2 = 2$ racionalan broj. Time je dokaz završen.

Dakle, dokazali smo tvrđenje o postojanju traženih brojeva, a da i dalje ne umemo eksplicitno da ih navedemo, jer zapravo ne znamo da li je $\sqrt{2}^{\sqrt{2}}$ racionalan ili iracionalan broj. Predstavnici tzv. *konstruktivističke matematike* odbacuju takve dokaze i zahtevaju da dokazi postojanja nekih objekata uključuju i mogućnost njihove efektivne konstrukcije. Kao logička osnova konstruktivističke matematike pojavljuje se *intuicionistička logika*. Intuicionistička logika odbacuje nekonstruktivne postupke dokazivanja koji su zasnovani na zakonu isključenja trećeg $A \vee \neg A$, kao i na zakonu dvostruke negacije $\neg \neg A \longrightarrow A$.

Dokazivanje svođenjem na kontradikciju je dopušteno samo za negativna tvrđenja: moguće je dokazati tvrđenje $\neg A$ tako što se dokaže da iz pretpostavke A sledi kontradikcija, ali nije dopušteno da se tvrđenje A dokaže tako što se iz pretpostavke $\neg A$ dokaže kontradikcija.

Do sada su svi dokazi bili u okviru intuicionističke logike. Dokazi u klasičnoj logici podrazumevaju uvođenje bar još jednog novog, klasičnog pravila. U sistemu Isabelle/HOL jedna mogućnost je da se koristi *pravilo klasične kontradikcije ccontr*: $(\neg P \Longrightarrow False) \Longrightarrow P$. U udžbenicima matematičke logike, to pravilo se zapisuje ovako:

$$\frac{\begin{array}{c}[\neg P]\\ \vdots\\ False\end{array}}{P}$$

Primetimo da smo za rad sa negacijom koristili smo pravilo uvođenja negacije *notI*: $(P \Longrightarrow False) \Longrightarrow \neg\ P$, koje je slično pravilu klasične kontradikcije. Ključna razlika je ta što se pravilom uvođenja negacije dokaz kontradikcijom primenjuje samo na negirane formule, što je u skladu sa intuicionističkom logikom i konstruktivnom matematikom. Otuda se u intuicionističkoj logici lako dokazuje naredna lema:

lemma $A \longrightarrow \neg\ \neg\ A$
apply (*rule impI*)
apply (*rule notI*)
apply (*erule notE*)
apply *assumption*
done

Međutim, u klasičnoj logici važi pravilo duple negacije:

Zadatak 3.34. $\neg\ \neg\ A \longrightarrow A$

lemma $\neg\ \neg\ A \longrightarrow A$
apply (*rule impI*)
— *1.* $\neg\ \neg\ A \Longrightarrow A$
— *Primetimo da ovde ne možemo uspešno primeniti pravilo notE (što će biti prikazano u narednom dokazu) zato što ono očekuje da iz* $\neg\ P$ *i* P *možemo da izvedemo bilo šta.* $\neg\ P$ *je ovde* $\neg\ \neg\ A$*, što znači da je* P *u stvari* $\neg\ A$*, i zato ono postaje preostali cilj koji se ne može dokazati.*
— *Primenjujemo pravilo ccontr.*
apply (*rule ccontr*)
— *1.* $[\![\neg\ \neg\ A; \neg\ A]\!] \Longrightarrow False$
— *Pa sada možemo da primenimo pravilo notE i jednostavno završavamo dokaz.*
apply (*erule notE*)
— *1.* $\neg\ A \Longrightarrow \neg\ A$
apply *assumption*
— *No subgoals*!
done

Dokaz, naravno, nije moguće izvršiti u intuicionističkoj logici.

lemma $\neg\ \neg\ A \longrightarrow A$
— *Jedino intuicionističko pravilo koje se može primeniti je impI*
apply (*rule impI*)

— *1.* $\neg\ \neg\ A \Longrightarrow A$
— *Jedino primenjivo pravilo je notE*
apply (*erule notE*)
— *1.* $\neg\ A$
— *Cilj koj je dobijen je jasno nedokaziv, pa odustajemo od dokaza.*
oops

Dokažimo i narednu klasičnu teoremu koja ne važi u intuicionističkoj logici:

Zadatak 3.35. $(\neg\ P \longrightarrow P) \longrightarrow P$

lemma $(\neg\ P \longrightarrow P) \longrightarrow P$
apply (*rule impI*)
— *1.* $\neg\ P \longrightarrow P \Longrightarrow P$
— *Ako bismo pokušali da primenimo impE dobili bi* $\neg\ P$ *kao novi cilj, što je nemoguće dokazati.*
— *Pa vidimo da je jedino primenjivo pravilo ccontr*
apply (*rule ccontr*)
— *1.* $[\![\neg\ P \longrightarrow P;\ \neg\ P]\!] \Longrightarrow False$
— *Sada pokušavamo sa impE*
apply (*erule impE*)
— *1.* $\neg\ P \Longrightarrow \neg\ P$
— *2.* $[\![\neg\ P;\ P]\!] \Longrightarrow False$
— *I vidimo da je u prvom cilju pretpostavka ista kao zaključak*
apply *assumption*
— *1.* $[\![\neg\ P;\ P]\!] \Longrightarrow False$
— *Primenjujemo notE*
apply (*erule notE*)
— *1.* $P \Longrightarrow P$
apply *assumption*
done

Od četiri smera de Morganovih zakona koji povezuju negaciju, disjunkciju i konjunkciju, tri važe i mogu se dokazati i u intuicionističkoj logici, dok je za dokaz narednog zakona potrebno upotrebiti i pravila klasične logike.

Zadatak 3.36. $\neg\ (A \wedge B) \longrightarrow \neg\ A \vee \neg\ B$

Često se nad disjunkcijom u zaključku primenjuje pravilo *ccontr*, što ćemo videti u narednom dokazu.

lemma $\neg\ (A \wedge B) \longrightarrow \neg\ A \vee \neg\ B$
apply (*rule impI*)
— *1.* $\neg\ (A \wedge B) \Longrightarrow \neg\ A \vee \neg\ B$

— Primena pravila ccontr će prebaciti zaključak u negativnom obliku u pretpostavke, i zajedno sa preostalim pretpostavkama pokušavamo da dokažemo kontradikciju.
apply (*rule ccontr*)
— 1. $[\![\neg\ (A \wedge B);\ \neg\ (\neg\ A \vee \neg\ B)]\!] \Longrightarrow$ *False*
— Sada imamo dve pretpostavke na koje možemo primeniti isto pravilo (u ovom slučaju notE), možemo da biramo da li ćemo to pravilo primeniti na prvu ili drugu pretpostavku. Međutim ovde nam odgovara da izaberemo prvu pretpostavku (i da ne upotrebimo naredbu back), zato što je u njoj konjunkcija - a ta formula se prebacuje u zaključak. Pogodnije je u zaključku raditi sa konjunkcijom nego sa disjunkcijom; Napomena: pogledajte šta se dešava kada stavite back
apply (*erule notE*)
— 1. $\neg\ (\neg\ A \vee \neg\ B) \Longrightarrow A \wedge B$
apply (*rule conjI*)
— Sada prvo primenjujemo pravilo uvođenja konjunkcije i dobijamo dva nova cilja:
— 1. $\neg\ (\neg\ A \vee \neg\ B) \Longrightarrow A$
— 2. $\neg\ (\neg\ A \vee \neg\ B) \Longrightarrow B$
— Sada dokazujemo prvi podcilj i ponovo primenjujemo pravilo ccontr.
apply (*rule ccontr*)
— 1. $[\![\neg\ (\neg\ A \vee \neg\ B);\ \neg\ A]\!] \Longrightarrow$ *False*
— 2. $\neg\ (\neg\ A \vee \neg\ B) \Longrightarrow B$
apply (*erule notE*)
— Sada primenjujemo pravilo notE na prvu pretpostavku jer nam u ovoj situaciji odgovara da imamo disjunkciju u zaključku.
— 1. $\neg\ A \Longrightarrow \neg\ A \vee \neg\ B$
— 2. $\neg\ (\neg\ A \vee \neg\ B) \Longrightarrow B$
apply (*rule disjI1*)
— 1. $\neg\ A \Longrightarrow \neg\ A$
— 2. $\neg\ (\neg\ A \vee \neg\ B) \Longrightarrow B$
apply *assumption*
— Dokazan je prvi podcilj i sada dokazujemo drugi podcilj.
— 1. $\neg\ (\neg\ A \vee \neg\ B) \Longrightarrow B$
— Ponovo primenjujemo pravilo ccontr i završavamo dokaz na sličan način kao malopre.
apply (*rule ccontr*)
— 1. $[\![\neg\ (\neg\ A \vee \neg\ B);\ \neg\ B]\!] \Longrightarrow$ *False*
apply (*erule notE*)
— 1. $\neg\ B \Longrightarrow \neg\ A \vee \neg\ B$
apply (*rule disjI2*)
— 1. $\neg\ B \Longrightarrow \neg\ B$
apply *assumption*
— No subgoals!
done

Zadatak 3.37. Pokazati klasični deo de-Morganovog zakona za kvantifikatore: $(\neg\ (\forall\ x.\ P\ x)) \longrightarrow (\exists\ x.\ \neg\ P\ x)$

Napomena: Slično kao i u slučaju iskazne logike, ovo je jedino od četiri de-Morganova pravila za kvantifikatore, koje se dokazuje uz pomoć principa klasične logike.

Intuitivno, ako važi pretpostavka $\neg\ (\forall\ x.\ P\ x)$, onda $P\ x$ nije tačno za svako x, odnosno postoji neko a za koje $P\ a$ nije tačno. Odnosno, za koje važi $\neg\ P\ a$. S obzirom na ovu činjenicu, možemo da zaključimo da postoji neko x za koje je $\neg\ P\ x$ tačno, tj. tačno je $\exists\ x.\ \neg\ P\ x$.

lemma *de-Morgan*: $(\neg\ (\forall\ x.\ P\ x)) \longrightarrow (\exists\ x.\ \neg\ P\ x)$
apply (*rule impI*)
— *1.* $\neg\ (\forall x.\ P\ x) \Longrightarrow \exists x.\ \neg\ P\ x$
— *Primenjujemo pravilo ccontr, pretpostavljamo suprotno, tj. prebacujemo negaciju zaključka na levu stranu sa ciljem da dokažemo kontradikciju.*
apply (*rule ccontr*)
— *1.* $[\![\neg\ (\forall x.\ P\ x);\ \nexists x.\ \neg\ P\ x]\!] \Longrightarrow False$
apply (*erule notE*)
— *1.* $[\![\nexists x.\ \neg\ P\ x]\!] \Longrightarrow \forall x.\ P\ x$
— *Primenjujemo pravilo notE da bismo eliminisali negaciju u prvoj pretpostavci. Nakon ovog pravila došli smo u situaciju da umesto polazne teoreme dokazujemo njenu kontrapoziciju (sa eliminisanom dvostrukom negacijom).*
apply (*rule allI*)
— *Dokaz nastavljamo na uobičajeni način, uvodeći univerzalni kvantifikator (tj. dokazujući tvrđenje za proizvoljno ali fiksirano x.*
— *1.* $\bigwedge x.\ \nexists x.\ \neg\ P\ x \Longrightarrow P\ x$
— *U nastavku ponovo kombinacijom pravila ccontr i notE razmenjujemo levu i desnu stranu i prelazimo na dokaz kontrapozicije.*
apply (*rule ccontr*)
apply (*erule notE*)
— *1.* $\bigwedge x.\ \neg\ P\ x \Longrightarrow \exists\ x.\ P\ x$
— *Dokaz je sada jednostavno završiti, jer je uvedeno proizvoljno x, svedok za* $\neg\ P\ x$.
apply (*rule-tac x=x* **in** *exI*)
apply *assumption*
done

3.7.2 Dodatni primeri

U klasičnoj logici, dokazati naredna tvrđenja:

Zadatak 3.38. $(\neg\ B \longrightarrow \neg\ A) \longrightarrow (A \longrightarrow B)$

lemma $(\neg\ B \longrightarrow \neg\ A) \longrightarrow (A \longrightarrow B)$
apply (*rule impI*)
— *1.* $\neg\ B \longrightarrow \neg\ A \Longrightarrow A \longrightarrow B$
apply (*rule impI*)
— *1.* $[\![\neg\ B \longrightarrow \neg\ A;\ A]\!] \Longrightarrow B$
apply (*rule ccontr*)
— *1.* $[\![\neg\ B \longrightarrow \neg\ A;\ A;\ \neg\ B]\!] \Longrightarrow False$
— *prebacuje zaključak sa desna na levo i pokušavamo da izvedemo kontradikciju*
apply (*erule impE*)
— *1.* $[\![A;\ \neg\ B]\!] \Longrightarrow \neg\ B$
— *2.* $[\![A;\ \neg\ B;\ \neg\ A]\!] \Longrightarrow False$
apply *assumption*
— *1.* $[\![A;\ \neg\ B;\ \neg\ A]\!] \Longrightarrow False$
apply (*erule notE*)
— *1.* $[\![A;\ \neg\ A]\!] \Longrightarrow B$
— *eliminacija negacije, ali treba da je primenimo na drugu negaciju, ne na* $\neg\ B$*, već na* $\neg\ A$
— *1.* $[\![A;\ \neg\ A]\!] \Longrightarrow B$
back
— *1.* $[\![A;\ \neg\ B]\!] \Longrightarrow A$
apply *assumption*
done

Zadatak 3.39. $(A \longrightarrow B) \longrightarrow (\neg\ A \vee B)$

lemma $(A \longrightarrow B) \longrightarrow (\neg\ A \vee B)$
apply (*rule impI*)
— *1.* $A \longrightarrow B \Longrightarrow \neg\ A \vee B$
apply (*rule ccontr*)
— *1.* $[\![A \longrightarrow B;\ \neg\ (\neg\ A \vee B)]\!] \Longrightarrow False$
apply (*erule impE*)
— *1.* $\neg\ (\neg\ A \vee B) \Longrightarrow A$
— *2.* $[\![\neg\ (\neg\ A \vee B);\ B]\!] \Longrightarrow False$
apply (*rule ccontr*)
— *1.* $[\![\neg\ (\neg\ A \vee B);\ \neg\ A]\!] \Longrightarrow False$
— *2.* $[\![\neg\ (\neg\ A \vee B);\ B]\!] \Longrightarrow False$
apply (*erule notE*)
— *1.* $\neg\ A \Longrightarrow \neg\ A \vee B$
— *2.* $[\![\neg\ (\neg\ A \vee B);\ B]\!] \Longrightarrow False$
apply (*rule disjI1*)
— *1.* $\neg\ A \Longrightarrow \neg\ A$
— *2.* $[\![\neg\ (\neg\ A \vee B);\ B]\!] \Longrightarrow False$
apply *assumption*

— *1.* $[\![\neg\ (\neg\ A \vee B);\ B]\!] \Longrightarrow False$
apply (*erule notE*)
— *1.* $B \Longrightarrow \neg\ A \vee B$
apply (*rule disjI2*)
— *1.* $B \Longrightarrow B$
apply *assumption*
done

Zadatak 3.40. $(\neg\ P \longrightarrow Q) \longleftrightarrow (\neg\ Q \longrightarrow P)$

lemma $(\neg\ P \longrightarrow Q) \longleftrightarrow (\neg\ Q \longrightarrow P)$
apply (*rule iffI*)
— *Dobijamo dva cilja; sada prvo primenjujemo pravilo uvođenja implikacije u prvom cilju.*
apply (*rule impI*)
— *Sada prebacujemo zaključak na levu stranu i desno dobijamo False.*
apply (*rule ccontr*)
— *Ovo nam odgovara zato što se sada* $\neg\ P$ *dodaje među pretpostavke, a već se pojavljuje kao leva strana implikacije.*
apply (*erule impE*)
apply (*assumption*)
apply (*erule notE*)
apply *assumption*
— *Sada smo zatvorili prvi cilj, pa slično sa drugim ciljom.*
apply (*rule impI*)
apply (*rule ccontr*)
— *Slično kao u prethodnoj primeni, odgovara nam da* $\neg\ Q$ *prebacimo u pretpostavke jer se već pojavljuje kao leva strana implikacije.*
apply (*erule impE*)
apply *assumption*
apply (*erule notE*)
apply *assumption*
done

Zadatak 3.41. $(\neg\ P \longrightarrow Q) \longleftrightarrow (\neg\ Q \longrightarrow P)$

lemma $(\neg\ P \longrightarrow Q) \longleftrightarrow (\neg\ Q \longrightarrow P)$
apply (*rule iffI*)
apply (*rule impI*)
apply (*rule ccontr*)
apply (*erule impE*)
apply *assumption*
apply (*erule notE*)

apply *assumption*
apply (*rule impI*)
apply (*rule ccontr*)
apply (*erule impE*)
apply *assumption*
apply (*erule notE*)
apply *assumption*
done

Zadatak 3.42. Pirsov zakon. Tipičan ne-intuicionistički primer koji će sigurno zahtevati neki oblik klasične kontradikcije. $((P \longrightarrow Q) \longrightarrow P) \longrightarrow P$

lemma $((P \longrightarrow Q) \longrightarrow P) \longrightarrow P$
apply (*rule impI*)
apply (*rule ccontr*)
— *Primena klasične kontradikcije, dodajemo negirani zaključak kao novu pretpostavku i pokušavamo da izvedemo kontradikciju.*
apply (*erule impE*)
apply (*rule impI*)
apply (*erule notE*)
apply *assumption*
apply (*erule notE*)
apply *assumption*
done

Aladin se nalazi ispred pećine u kojoj su dva kovčega. U svakom od kovčega se nalazi ili blago ili smrtonosna zamka. Na kovčegu A piše da se bar u jednom od dva kovčega nalazi blago. Na kovčegu B piše da se u kovčegu A nalazi smrtonosna zamka. Poznato je da su ili oba natpisa tačna ili da nijedan od njih nije tačan. Odrediti sadržaj kovčega i dokazati odgovor.

Ako nijedan od natpisa ne bi bio tačan, tada nijedan od kovčega ne bi mogao da sadrži blago (jer je natpis A netačan), dok bi kovčeg A morao da sadrži blago (pošto je natpis B netačan). Ovo je kontradiktorno, pa je nemoguće da su oba natpisa netačna. Dakle, oba natpisa moraju biti tačna. Tada kovčeg A sadrži smrtonosnu zamku, dok kovčeg B sadrži blago.

Formalizujmo ova tvrđenja u iskaznoj logici. Neka iskazna promenljiva bA označava da kovčeg A sadrži blago, a iskazna promenljiva bB da kovčeg B sadrži blago (tada $\neg\ bA$ označava da je u kovčegu A smrtonosna zamka, a $\neg\ bB$ da je u kovčegu B smrtonosna zamka. Neka promenljiva nA označava da je natpis na kovčegu A tačan, a promenljiva nB da je natpis na kovčegu B tačan.

Zadatak 3.43. $(nA \longleftrightarrow bA \vee bB) \wedge (nB \longleftrightarrow \neg\ bA) \wedge ((nA \wedge nB) \vee (\neg\ nA \wedge \neg\ nB)) \longrightarrow \neg\ bA \wedge bB$

lemma
$(nA \longleftrightarrow bA \vee bB) \wedge (nB \longleftrightarrow \neg\ bA) \wedge ((nA \wedge nB) \vee (\neg\ nA \wedge \neg\ nB)) \longrightarrow \neg\ bA \wedge bB$
— *Primenjujemo očigledna pravila*
apply (*rule impI*)
apply (*erule conjE*)+
apply (*erule iffE*)+
— *Razmatramo dva slučaja*
apply (*erule disjE*)
— *U prvom slučaju su oba natpisa tačna*
apply (*erule conjE*)
— *Iz pretpostavki izvodimo da važi* $\neg\ bA$ *i da važi* $bA \vee bB$
apply (*erule impE*)
apply *assumption*
apply (*erule impE*)
back
apply *assumption*

— *Posebno dokazujemo da važi* $\neg\ bA$ *i posebno* bB
apply (*rule conjI*)
apply *assumption*
— *Pošto znamo da važi* $bA \vee bB$ *analiziramo slučaj da važi* bA *i* bB
apply (*erule disjE*)
apply (*erule notE*)
apply *assumption*
apply *assumption*

— *U ovom slučaju nijedan od natpisa nije istinit. Dokaćemo da je to kontradiktorno*
apply (*erule conjE*)
— *Dokazaćemo prvo da iz pretpostavki sledi da blago mora biti bar u jednom kovčegu*
apply (*erule impE*) **back**
— *Pretpostavljamo suprotno (da blago nije ni u jednom kovčegu)*
apply (*rule ccontr*)
— *Odatle sledi da blago nije u kovčegu A*
apply (*erule impE*) **back back**
apply (*rule notI*)
apply (*erule notE*) **back back**
apply (*rule disjI1*)
apply *assumption*
— *Stoga je natpis B istinit, što je kontradikcija sa tim da nijedan od natpisa nije*

istinit.
apply (*erule notE*) **back**
apply *assumption*
— *Pošto je blago bar u jednom kovčegu, natpis A je istinit, što je kontradikcija sa tim da nijedan od natpisa nije istinit.*
apply (*erule notE*)
apply *assumption*
done

3.7.3 Pravilo classical

Još jedno pravilo klasične logike koje možemo koristiti u sistemu Isabelle/HOL je *classical*: $(\neg\ ?P \Longrightarrow\ ?P) \Longrightarrow\ ?P$. Ovo pravilo podrazumeva da u dokazu teoreme uvek možemo pretpostaviti negaciju zaključka teoreme dok pokušavamo da izvedemo upravo taj zaključak. Pravilo je slično pravilu svođenja na kontradikciju *ccontr*. Jedina razlika je to što se kod pravila *ccontr* iz negacije zaključka izvodi kontradikcija *False*, a kod pravila *classical* izvodi zaključak teoreme.

Dokažimo zakon isključenja trećeg korišćenjem pravila *classical* (primetimo da njega ne bismo mogli da dokažemo samo korišćenjem pravila *ccontr* i pravilima uvođenja i eliminacije veznika).

Zadatak 3.44. $P \vee \neg\ P$

lemma $P \vee \neg\ P$
— *Da bi smo dokazali formulu A, dokazaćemo formulu* $\neg\ A \Rightarrow A$*, pa primenjujemo pravilo classical.*
apply (*rule classical*)
— *1.* $\neg\ (P \vee \neg\ P) \Longrightarrow P \vee \neg\ P$
— *Pošto je situacija simetrična, biramo da u cilju zadržimo prvi disjunkt, tj. P.*
apply (*rule disjI1*)
— *1.* $\neg\ (P \vee \neg\ P) \Longrightarrow P$
— *Dokaz možemo nastaviti običnim svođenjem na kontradikciju, pa primenjujemo pravilo ccontr (naravno, bilo bi sasvim u redu i da se ponovo primeni pravilo classical.*
apply (*rule ccontr*)
— *1.* $[\![\neg\ (P \vee \neg\ P); \neg\ P]\!] \Longrightarrow False$
apply (*erule notE*)
— *1.* $\neg\ P \Longrightarrow P \vee \neg\ P$
apply (*rule disjI2*)
— *1.* $\neg\ P \Longrightarrow \neg\ P$
apply *assumption*

done

end

3.7.4 Dokazi po slučajevima

theory *Cas5-vezbe*
imports *Main*
begin

Dokazi po slučajevima se koriste kada je potrebno izvršiti grananje po nekom uslovu u toku samog dokaza. Za navođenje uslova po kom se grananje vrši koristi se metod *cases*. U ovom poglavlju neće biti navedeno puno primera, iz jednostavnog razloga što se apply-skript dokazi teško prate i kada je reč o linearnim dokazima. Otuda je i za očekivati da će dokazi sa grananjem biti znatno teži. Prikazaćemo dokaze dve leme.

Zadatak 3.45. $(A \longleftrightarrow (A \longleftrightarrow B)) \longrightarrow B$

Prvo će biti naveden dokaz ove leme bez objašnjenja, a zatim isti dokaz detaljno iskomentarisan sa stanjima kroz koje prolazi dokazivač u toku dokazivanja.

lemma *no-one-admits-knave-apply*:
$(A \longleftrightarrow (A \longleftrightarrow B)) \longrightarrow B$
apply (*rule impI*)
apply (*cases A*)
— *Prva grana.*
apply (*erule iffE*)
apply (*erule impE*)
apply *assumption*
apply (*erule iffE*)
apply (*erule impE*)
back
apply *assumption*
apply *assumption*
— *Druga grana.*
apply (*rule ccontr*)
apply (*erule iffE*)
apply (*erule impE*)
back
apply (*rule iffI*)
apply (*erule notE*)
apply *assumption*
apply (*erule notE*)
back
apply *assumption*
apply (*erule notE*)
apply *assumption*

done

Detaljno objašnjen dokaz teoreme:

lemma *no-one-admits-knave-apply-detaljno*:
$(A \longleftrightarrow (A \longleftrightarrow B)) \longrightarrow B$
apply (*rule impI*)
— *1.* $A = (A = B) \Longrightarrow B$
— *Odmah na početku dokaza, granamo ovaj dokaz po slučajevima da li je A tačno ili netačno. Kada se samo jedan term koristi u okviru metoda cases nema potrebe koristiti navodnike (inače su neophodni, što ćemo videti u narednom primeru).*
— *Ovo će generisati dva nova cilja, u prvom se u skup pretpostavki dodaje nova pretpostavka da važi A, a u drugom se u skup pretpostavki dodaje pretpostavka da važi* $\neg A$.
apply (*cases A*)
— *1.* $[\![A = (A = B);\ A]\!] \Longrightarrow B$
— *2.* $[\![A = (A = B);\ \neg A]\!] \Longrightarrow B$

— *Sada dokazujemo prvi cilj.*
— *U pretpostavkama imamo ekvivalenciju pa primenjujemo pravilo iffE koje eliminiše ekvivalenciju i dodaje u skup pretpostavki dve implikacije.*
— *Primetimo da zagrade ne stoje oko jednakosti sada, zato što jednakost odnosno ekvivalencija ima veći prioritet u odnosu na implikaciju.*
apply (*erule iffE*)
— *1.* $[\![A;\ A \longrightarrow A = B;\ A = B \longrightarrow A]\!] \Longrightarrow B$
— *2.* $[\![A = (A = B);\ \neg A]\!] \Longrightarrow B$
— *Sada eliminišemo prvu implikaciju (što nam odgovara jer se njena pretpostavka A već nalazi u spisku pretpostavci tekućeg cilja).*
apply (*erule impE*)
— *1.* $[\![A;\ A = B \longrightarrow A]\!] \Longrightarrow A$
— *2.* $[\![A;\ A = B \longrightarrow A;\ A = B]\!] \Longrightarrow B$
— *3.* $[\![A = (A = B);\ \neg A]\!] \Longrightarrow B$
apply *assumption*
— *1.* $[\![A;\ A = B \longrightarrow A;\ A = B]\!] \Longrightarrow B$
— *2.* $[\![A = (A = B);\ \neg A]\!] \Longrightarrow B$
— *Sada elimišemo ekvivalenciju* $A = B$ *što generiše dve nove implikacije koje se dodaju u pretpostavke.*
apply (*erule iffE*)
— *1.* $[\![A;\ A = B \longrightarrow A;\ A \longrightarrow B;\ B \longrightarrow A]\!] \Longrightarrow B$
— *2.* $[\![A = (A = B);\ \neg A]\!] \Longrightarrow B$
— *Sada imamo tri implikacije u pretpostavkama i želimo da eliminišemo drugu implikaciju (jer formulu A imamo među ostalim pretpostavkama). Pa odmah planiramo da pozovemo naredbu back nakon ove naredbe.*
apply (*erule impE*)

— *1.* $[\![A;\ A \longrightarrow B;\ B \longrightarrow A]\!] \Longrightarrow A = B$
— *2.* $[\![A;\ A \longrightarrow B;\ B \longrightarrow A;\ A]\!] \Longrightarrow B$
— *3.* $[\![A = (A = B);\ \neg\ A]\!] \Longrightarrow B$
back
— *1.* $[\![A;\ A = B \longrightarrow A;\ B \longrightarrow A]\!] \Longrightarrow A$
— *2.* $[\![A;\ A = B \longrightarrow A;\ B \longrightarrow A;\ B]\!] \Longrightarrow B$
— *3.* $[\![A = (A = B);\ \neg\ A]\!] \Longrightarrow B$
apply *assumption*
— *1.* $[\![A;\ A = B \longrightarrow A;\ B \longrightarrow A;\ B]\!] \Longrightarrow B$
— *2.* $[\![A = (A = B);\ \neg\ A]\!] \Longrightarrow B$
apply *assumption*
— *Dokazali smo prvi cilj, odnosno prvi slučaj kada je A tačno.*

— *Sada dokazujemo drugi cilj.*
— *1.* $[\![A = (A = B);\ \neg\ A]\!] \Longrightarrow B$
— *Sada primenjujemo pravilo ccontr, odnosno pretpostavimo suprotno od tekućeg cilja i dokažimo da onda dobijamo kontradikciju.*
apply (*rule ccontr*)
— *1.* $[\![A = (A = B);\ \neg\ A;\ \neg\ B]\!] \Longrightarrow$ *False*
— *Sada prvo eliminišemo ekvivalenciju iz pretpostavki, čime dobijamo dve nove implikacije.*
apply (*erule iffE*)
— *1.* $[\![\neg\ A;\ \neg\ B;\ A \longrightarrow A = B;\ A = B \longrightarrow A]\!] \Longrightarrow$ *False*
— *Pošto prva implikacija sledi iz A, a u pretpostavkama imamo da važi* $\neg\ A$*, možemo da zaključimo da nam ne odgovara da eliminišemo prvu implikaciju pa pokušavamo sa drugom i odmah planiramo da pozovemo naredbu back nakon ove naredbe.*
apply (*erule impE*)
— *1.* $[\![\neg\ A;\ \neg\ B;\ A = B \longrightarrow A]\!] \Longrightarrow A$
— *2.* $[\![\neg\ A;\ \neg\ B;\ A \longrightarrow A = B;\ A]\!] \Longrightarrow$ *False*
back
— *1.* $[\![\neg\ A;\ \neg\ B;\ A \longrightarrow A = B]\!] \Longrightarrow A = B$
— *2.* $[\![\neg\ A;\ \neg\ B;\ A \longrightarrow A = B;\ A]\!] \Longrightarrow$ *False*
— *Sada primenjujemo pravilo uvođenja ekvivalencije na* $A = B$ *i dobijamo dva nova cilja.*
apply (*rule iffI*)
— *1.* $[\![\neg\ A;\ \neg\ B;\ A \longrightarrow A = B;\ A]\!] \Longrightarrow B$
— *2.* $[\![\neg\ A;\ \neg\ B;\ A \longrightarrow A = B;\ B]\!] \Longrightarrow A$
— *3.* $[\![\neg\ A;\ \neg\ B;\ A \longrightarrow A = B;\ A]\!] \Longrightarrow$ *False*
— *Pošto u pretpostavkama imamo* $\neg\ A$ *i A, odgovara nam da pokušamo da eliminišemo prvu negaciju.*
apply (*erule notE*)
— *1.* $[\![\neg\ B;\ A \longrightarrow A = B;\ A]\!] \Longrightarrow A$

— *2.* $[\![\neg\ A;\ \neg\ B;\ A \longrightarrow A = B;\ B]\!] \Longrightarrow A$
— *3.* $[\![\neg\ A;\ \neg\ B;\ A \longrightarrow A = B;\ A]\!] \Longrightarrow \mathit{False}$
apply *assumption*
— *1.* $[\![\neg\ A;\ \neg\ B;\ A \longrightarrow A = B;\ B]\!] \Longrightarrow A$
— *2.* $[\![\neg\ A;\ \neg\ B;\ A \longrightarrow A = B;\ A]\!] \Longrightarrow \mathit{False}$
— *Pošto u pretpostavkama imamo* $\neg\ B$ *i* B*, odgovara nam da pokušamo da eliminišemo drugu negaciju pa planiramo da pozovemo back nakon naredne naredbe.*
apply (*erule notE*)
— *1.* $[\![\neg\ B;\ A \longrightarrow A = B;\ B]\!] \Longrightarrow A$
— *2.* $[\![\neg\ A;\ \neg\ B;\ A \longrightarrow A = B;\ A]\!] \Longrightarrow \mathit{False}$
back
— *1.* $[\![\neg\ A;\ A \longrightarrow A = B;\ B]\!] \Longrightarrow B$
— *2.* $[\![\neg\ A;\ \neg\ B;\ A \longrightarrow A = B;\ A]\!] \Longrightarrow \mathit{False}$
apply *assumption*
— *1.* $[\![\neg\ A;\ \neg\ B;\ A \longrightarrow A = B;\ A]\!] \Longrightarrow \mathit{False}$
— *Pošto u pretpostavkama imamo* $\neg\ A$ *i* A*, odgovara nam da pokušamo da eliminišemo prvu negaciju.*
apply (*erule notE*)
— *1.* $[\![\neg\ B;\ A \longrightarrow A = B;\ A]\!] \Longrightarrow A$
apply *assumption*
— *No subgoals*!
done

Zadatak 3.46. Paradoks pijanca: postoji osoba za koju važi, ako je on pijanac onda su i svi ostali pijanci.

Pokazaćemo nekoliko različitih dokaza ovog tvrđenja.

Direktan dokaz: U svim dokazima granamo prema zaključku teoreme. Ponovo koristimo metod *cases* ali sada formulu po kojom granamo navodimo između dvostrukih navodnika.

Prvo navodimo dokaz bez objašnjenja, pa nakon toga detaljno objašnjen dokaz sa stanjima kroz koja prolazi dokazivač.

lemma *Drinker's-Principle1-bez-objasnjenja*:
$\exists\ x.\ (\mathit{drunk}\ x \longrightarrow (\forall\ x.\ \mathit{drunk}\ x))$
apply (*cases* $\forall\ x.\ \mathit{drunk}\ x$)
— prva grana
apply (*rule exI*)
apply (*rule impI*)
apply *assumption*
— druga grana
apply (*rule ccontr*)

apply (*erule notE*)
apply (*rule allI*)
apply (*rule ccontr*)
apply (*erule notE*)
apply (*rule-tac* $x = x$ **in** *exI*)
apply (*rule impI*)
apply (*erule notE*)
apply *assumption*
done

Detaljno objašnjen dokaz teoreme:

lemma *Drinker's-Principle1*:
$\exists\ x.\ (drunk\ x \longrightarrow (\forall\ x.\ drunk\ x))$
— *Prvo granamo prema tome da li zaključak važi ili ne, primenjujemo metod cases i dobijamo dve grane, odnosno dva podcilja.*
apply (*cases* $\forall\ x.\ drunk\ x$)
— *1.* $\forall x.\ drunk\ x \Longrightarrow \exists x.\ drunk\ x \longrightarrow (\forall x.\ drunk\ x)$
— *2.* $\neg\ (\forall x.\ drunk\ x) \Longrightarrow \exists x.\ drunk\ x \longrightarrow (\forall x.\ drunk\ x)$
— *Sada dokazujemo prvu granu. Moramo primetiti da nam nije bitan svedok tako da možemo da primenimo pravilo exI bez instanciranja promenljive.*
apply (*rule exI*)
— *1.* $\forall x.\ drunk\ x \Longrightarrow drunk\ ?x1 \longrightarrow (\forall x.\ drunk\ x)$
— *2.* $\neg\ (\forall x.\ drunk\ x) \Longrightarrow \exists x.\ drunk\ x \longrightarrow (\forall x.\ drunk\ x)$
— *Sada primenjujemo pravilo uvođenja implikacije u zaključak.*
apply (*rule impI*)
— *1.* $[\![\forall x.\ drunk\ x;\ drunk\ ?x1]\!] \Longrightarrow \forall x.\ drunk\ x$
— *2.* $\neg\ (\forall x.\ drunk\ x) \Longrightarrow \exists x.\ drunk\ x \longrightarrow (\forall x.\ drunk\ x)$
— *I lako dokazujemo prvu granu.*
apply *assumption*

— *Sada dokazujemo drugu granu.*
— *1.* $\neg\ (\forall x.\ drunk\ x) \Longrightarrow \exists x.\ drunk\ x \longrightarrow (\forall x.\ drunk\ x)$
— *Prvo primenjujemo klasičnu kontradikciju, odnosno negaciju zaključka prebacujemo u pretpostavke i pokušavamo da izvedemo kontradikciju.*
apply (*rule ccontr*)
— *1.* $[\![\neg\ (\forall x.\ drunk\ x);\ \nexists x.\ drunk\ x \longrightarrow (\forall x.\ drunk\ x)]\!] \Longrightarrow False$
— *Sada eliminišemo negaciju iz pretpostavki.*
apply (*erule notE*)
— *1.* $\nexists x.\ drunk\ x \longrightarrow (\forall x.\ drunk\ x) \Longrightarrow \forall x.\ drunk\ x$
— *Pa univerzalni kvantifikator iz zaključka.*
apply (*rule allI*)
— *1.* $\bigwedge x.\ \nexists x.\ drunk\ x \longrightarrow (\forall x.\ drunk\ x) \Longrightarrow drunk\ x$
— *Ponovo primenjujemo pravilo klasične kontradikcije.*

apply (*rule ccontr*)
— *1.* $\bigwedge x.\ [\![\nexists x.\ drunk\ x \longrightarrow (\forall x.\ drunk\ x);\ \neg\ drunk\ x]\!] \Longrightarrow False$
— *Eliminišemo negaciju iz pretpostavki.*
apply (*erule notE*)
— *1.* $\bigwedge x.\ \neg\ drunk\ x \Longrightarrow \exists x.\ drunk\ x \longrightarrow (\forall x.\ drunk\ x)$
— *Sada instanciramo egzistencijalni kvantifikator u zaključku promenljivom x koju smo već uveli.*
apply (*rule-tac* $x = x$ **in** *exI*)
— *1.* $\bigwedge x.\ \neg\ drunk\ x \Longrightarrow drunk\ x \longrightarrow (\forall x.\ drunk\ x)$
— *Sada primenjujemo pravilo uvođenja implikacije u zaključku.*
apply (*rule impI*)
— *1.* $\bigwedge x.\ [\![\neg\ drunk\ x;\ drunk\ x]\!] \Longrightarrow \forall x.\ drunk\ x$
— *Sada eliminišemo negaciju iz pretpostavki.*
apply (*erule notE*)
— *1.* $\bigwedge x.\ drunk\ x \Longrightarrow drunk\ x$
apply *assumption*
— *No subgoals*!
done

Dokaz uz korišćenje pomoćne leme: U naredna dva dokaza paradoksa pijanca koristi se klasični deo de-Morganovog zakona koji smo već dokazali ali ga navodimo ponovo.

lemma *de-Morgan*: $(\neg\ (\forall\ x.\ P\ x)) \longrightarrow (\exists\ x.\ \neg\ P\ x)$
apply (*rule impI*)
apply (*rule classical*)
apply (*erule notE*)
apply (*rule allI*)
apply (*rule classical*)
apply (*erule notE*)
apply (*rule-tac* $x = x$ **in** *exI*)
apply *assumption*
done

Prikazaćemo dve verzije dokaza uz pomoć pravila prirodne dedukcije koje koriste ovu lemu.

Korišćenje pomoćne leme u dokazu: Moguće je koristiti i pomoćne leme u apply-skript dokazima, na način koji je veoma sličan primeni pravila prirodne dedukcije.

- `apply (rule` *theorem*`)` - primenjuje se ako se zaključak teoreme poklapa sa zaključkom tekućeg cilja.

- `apply (erule` *theorem*`)` - primenjuje se ako se zaključak teoreme poklapa sa zaključkom tekućeg cilja i prva pretpostavka se poklapa sa pretpostavkom tekućeg cilja.

- `apply (frule` *theorem*`)` - primenjuje se kada se prva pretpostavka teoreme poklapa sa pretpostavkom tekućeg cilja.

- `apply (drule` *theorem*`)` - slično kao pravilo *frule* s tim što se pretpostavka koja se koristi briše.

Naredba [*rule-format*]**:** Naredba *rule-format* naglašava Isabelle/HOL da se u tvrđenju leme svako $\longrightarrow$ zameni sa $\Longrightarrow$ pre čuvanja ili primene dokazane leme. Može se koristiti ili prilikom pozivanja određene leme (u dokazu druge leme - kao što će biti prikazano u nastavku teksta) ili se može navesti odmah nakon definisanja same leme (što čitalac može pokušati sam). Ovakva transformacija leme olakšava njenu primenu u dokazima drugih lema.

Dodavanje među-cilja, metod *subgoal-tac***:** Metod *subgoal-tac* se koristi da bismo u dokazu dodali novu pretpostavku koju planiramo da koristimo u dokazu konačnog cilja. Ovo nam omogućava da koristimo pretpostavku pre nego što je dokažemo, i da se ovaj međukorak (odnosno pretpostavka koja je korišćena) dokazuje tek nakon dokazivanja konačnog cilja. Ovaj metod generiše dva cilja: prvi cilj nastaje od polaznog cilja čiji je spisak pretpostavki proširen novom pretpostavkom, drugi cilj je da se iz prethodnih pretpostavki dokaže nova pretpostavka.

Videćemo i razliku između primene lema *de-Morgan* (zapisane sa $\longrightarrow$) i *de-Morgan1* (zapisane sa $\Longrightarrow$).

lemma *Drinker's-Principle2*:
$\exists\ x.\ (drunk\ x \longrightarrow (\forall\ x.\ drunk\ x))$
apply (*cases* $\forall\ x.\ drunk\ x$)
— *Prva grana se dokazuje na isti način*
apply (*rule exI*)
apply (*rule impI*)
apply *assumption*
— *Druga grana se razlikuje!*
— *Koristimo pravilo subgoal-tac i dodajemo novu pretpostavku. Ovo generiše dva cilja..*
apply (*subgoal-tac* $\exists\ x.\ \neg\ drunk\ x$)
— *1.* $[\![\neg\ (\forall x.\ drunk\ x); \exists x.\ \neg\ drunk\ x]\!] \Longrightarrow \exists x.\ drunk\ x \longrightarrow (\forall x.\ drunk\ x)$
— *2.* $\neg\ (\forall x.\ drunk\ x) \Longrightarrow \exists x.\ \neg\ drunk\ x$
— *Sada iz pretpostavki (iz nove pretpostavke) eliminišemo egzistencijalni kvantifikator.*

apply (*erule exE*)
— *1.* $\bigwedge x.\ [\![\neg\ (\forall\, x.\ drunk\ x);\ \neg\ drunk\ x]\!] \Longrightarrow \exists\, x.\ drunk\ x \longrightarrow (\forall\, x.\ drunk\ x)$
— *2.* $\neg\ (\forall\, x.\ drunk\ x) \Longrightarrow \exists\, x.\ \neg\ drunk\ x$
— *Sada pokušavamo sa tim istim x primenimo pravilo uvođenja egzistencijalnog kvantifikatora u zaključku.*
apply (*rule-tac* $x = x$ **in** *exI*)
— *1.* $\bigwedge x.\ [\![\neg\ (\forall\, x.\ drunk\ x);\ \neg\ drunk\ x]\!] \Longrightarrow drunk\ x \longrightarrow (\forall\, x.\ drunk\ x)$
— *2.* $\neg\ (\forall\, x.\ drunk\ x) \Longrightarrow \exists\, x.\ \neg\ drunk\ x$
— *Sada primenjujemo pravilo uvođenja implikacije u zaključku, čime se pretpostavka iz zaključka prebacuje u pretpostavke samog cilja.*
apply (*rule impI*)
— *1.* $\bigwedge x.\ [\![\neg\ (\forall\, x.\ drunk\ x);\ \neg\ drunk\ x;\ drunk\ x]\!] \Longrightarrow \forall\, x.\ drunk\ x$
— *2.* $\neg\ (\forall\, x.\ drunk\ x) \Longrightarrow \exists\, x.\ \neg\ drunk\ x$
— *Sada u pretpostavkama imamo* $\neg$ *drunk x i drunk x, pa želimo da primenimo pravilo notE upravo na* $\neg$ *drunk x (što je druga negacija u pretpostavkama). Zbog toga odmah planiramo da nakon primene pravila notE pozovemo naredbu back.*
apply (*erule notE*)
— *1.* $\bigwedge x.\ [\![\neg\ drunk\ x;\ drunk\ x]\!] \Longrightarrow \forall\, x.\ drunk\ x$
— *2.* $\neg\ (\forall\, x.\ drunk\ x) \Longrightarrow \exists\, x.\ \neg\ drunk\ x$
back
— *1.* $\bigwedge x.\ [\![\neg\ (\forall\, x.\ drunk\ x);\ drunk\ x]\!] \Longrightarrow drunk\ x$
— *2.* $\neg\ (\forall\, x.\ drunk\ x) \Longrightarrow \exists\, x.\ \neg\ drunk\ x$
apply *assumption*
— *1.* $\neg\ (\forall\, x.\ drunk\ x) \Longrightarrow \exists\, x.\ \neg\ drunk\ x$
— *Sada je dokazan konačni cilj. Ostaje nam da dokažemo međukorak direktnom primenom leme de-Morgan nakon što je transformišemo u odgovarajući oblik naredbom* [*rule-format*]. Pošto se u zaključku teoreme nalazi upravo zaključak leme *de-Morgan* pozivamo je sa metodom *rule*.
apply (*rule de-Morgan*[*rule-format*])
— *1.* $\neg\ (\forall\, x.\ drunk\ x) \Longrightarrow \neg\ (\forall\, x.\ drunk\ x)$
apply *assumption*
— *No subgoals*!
done

Drugi način: U narednom dokazu koristimo metod *frule*. Ono je nalik metodu *erule* (označava rezonovanje unapred, *forward-resolution*) s tim što se pretpostavka na koju je pravilo primenjeno ne briše iz skupa pretpostavki. Primenjuje se za korišćenje pomoćne leme, u situaciji kada se prva pretpostavka pomoćne leme može unifikovati sa pretpostavkom tekućeg cilja.

lemma *Drinker's-Principle3*:
$\exists\ x.\ (drunk\ x \longrightarrow (\forall\ x.\ drunk\ x))$

apply (*cases* $\forall$ *x. drunk x*)
apply (*rule exI*)
apply (*rule impI*)
apply *assumption*
— *Druga grana:*
— *Sada primenjujemo ranije dokazanu lemu uz pomoć metoda frule, zato što se pretpostavka teoreme poklapa sa pretpostavkom tekućeg cilja. Isto kao i ranije, potrebno je prvo transformisati lemu komandom* [*rule-format*].
apply (*frule de-Morgan*[*rule-format*])
— *Ostatak dokaza se izvodi slično kao u prethodnom dokazu, nakon koraka koji koristi naredbu subgoal-tac.*
— *1.* $[\![\neg\ (\forall x.\ drunk\ x);\ \exists x.\ \neg\ drunk\ x]\!] \Longrightarrow \exists x.\ drunk\ x \longrightarrow (\forall x.\ drunk\ x)$
apply (*erule exE*)
— *1.* $\bigwedge x.\ [\![\neg\ (\forall x.\ drunk\ x);\ \neg\ drunk\ x]\!] \Longrightarrow \exists x.\ drunk\ x \longrightarrow (\forall x.\ drunk\ x)$
apply (*rule-tac x = x* **in** *exI*)
— *1.* $\bigwedge x.\ [\![\neg\ (\forall x.\ drunk\ x);\ \neg\ drunk\ x]\!] \Longrightarrow drunk\ x \longrightarrow (\forall x.\ drunk\ x)$
apply (*rule impI*)
— *1.* $\bigwedge x.\ [\![\neg\ (\forall x.\ drunk\ x);\ \neg\ drunk\ x;\ drunk\ x]\!] \Longrightarrow \forall x.\ drunk\ x$
apply (*erule notE*)
— *1.* $\bigwedge x.\ [\![\neg\ drunk\ x;\ drunk\ x]\!] \Longrightarrow \forall x.\ drunk\ x$
back
— *1.* $\bigwedge x.\ [\![\neg\ (\forall x.\ drunk\ x);\ drunk\ x]\!] \Longrightarrow drunk\ x$
apply *assumption*
— *No subgoals*!
done

end

4

Programski jezik Isar i struktuirani dokazi

4.1 Osnove jezika Isar

theory *Cas6-vezbe*
imports *Main*

begin

declare [[*quick-and-dirty*]]

Programski jezik Isar nudi mogućnost pisanja dokaza na način koji podseća na dokaze u matematičkim udžbenicima. Tako generisani dokazi su čitljivi kako za računar tako i za čoveka.

Izražajnost jezika Isar je mnogo veća nego što će biti prikazano u ovoj knjizi. Detaljan opis jezika Isar i njegove sintakse se može naći u [1] [2].

U ovom poglavlju ćemo prvo objasniti sintaksu Isar dokaza, a zatim ćemo prikazati šablone za dokazivanje nekih osnovnih matematičkih tvrđenja. U slučaju jednostavnijih tvrđenja, automatski dokazivači (na primer metod *auto*) mogu dokazati zadata tvrđenja. Međutim, u ovom poglavlju nam nije cilj samo provera da li je neko tvrđenje teorema ili ne, već želimo da na jednostavnijim primerima naučimo osnove jezika Isar.

U svakom koraku dokaza se jasno navodi koji je trenutni cilj. Otuda su dve glavne karakteristike ovog jezika:

- Njegova struktura je razgranata (nije linearna).
- Dokazi su čitljivi i bez interpretatora.

Isar dokaz može biti ili trivijalan (kada se završava u jednom koraku) korišćenjem naredbe `by` nakon koje sledi automatski metod koji se koristi (na primer *auto* ili *simp*), ili može imati više koraka koji su raspoređeni unutar *proof* — *qed* bloka. U tom slučaju korisnik može (ako za tim ima potrebe) precizirati koji metod dokazivanja želi da koristi (kao što se radi u dokazima po slučajevima, dokazima indukcijom i slično).

4.2 Primer Isar dokaza

U ovom poglavlju ćemo prikazati nekoliko načina na koje možemo da zapišemo jedan jednostavan matematički dokaz Kantorove teoreme.

[1] Markus Wenzel PhD Thesis
[2] Isar reference manual

Kantorova teorema tvrdi da ne postoji surjektivna funkcija koja slika skup u njegov skup podskupova.

U opštem obliku, Isar dokaz tvrđenja $formula_0 \Longrightarrow formula_{n+1}$ (pod pretpostavkom da je svaki od navedenih koraka uspešno izvršen) bi izgledao ovako:

proof
  assume $formula_0$
  have $formula_1$ by *simp*
  ⋮
  have $formula_n$ by *blast*
  show $formula_{n+1}$ by ...
qed

Neke od osnovnih naredbi jezika Isar:

- Naredba *assume* navodi pretpostavke glavnog tvrđenja.

- Naredba *show* navodi zaključak glavnog tvrđenja.

- Naredba *proof* poziva neki od metoda prirodne dedukcije, u skladu sa tekućim tvrđenjem koje se dokazuje.

- Naredba *have* se koristi za izvođenje međukoraka (za povezivanje pretpostavke tvrđenja sa njenim zaključkom).

- Opciono nakon naredbe *have* može se imenovati novouvedena činjenica korišćenjem labele (simbol ispred dvotačke) koja se navodi pre same činjenice.

- Opciono, u svakom koraku može se javiti i *from* klauza koja nam omogućava da navedemo činjenice koje će biti korišćene u tekućem koraku.

- Opciono, moguće je uvesti nove promenljive naredbom *fix*, ona uvodi $\bigwedge$-kvantifikovane promenljive.

Jedan kratak Isar dokaz koji koristi ove naredbe može izgledati ovako:

lemma $\neg$ *surj*($f :: 'a \Rightarrow 'a\ set$)
proof
  assume *0*: *surj f*
  from *0* **have** *1*: $\forall A.\ \exists a.\ A = f\ a$ **by**(*simp add*: *surj-def*)
  from *1* **have** *2*: $\exists a.\ \{x.\ x \notin f\ x\} = f\ a$ **by** *blast*
  from *2* **show** *False* **by** *blast*
qed

4.2.1 Dodatne opcije *this*, *then*, *hence*, *thus*, *with*, *using*

U prethodnom dokazu su korišćene labele za imenovanje prethodnih činjenica. Kako se često u matematičkim dokazima dešava da se pozivamo na činjenicu dokazanu u (neposredno) prethodnom koraku, uveden je predefinisan naziv *this* za poslednju činjenicu izvedenu u prethodnom koraku. Sada dokaz možemo zapisati i na sledeći način:

proof
  assume *surj f*
  from *this* **have** $\exists\, a.\ \{x.\ x \notin f\ x\} = f\ a$ **by**(*auto simp*: *surj-def*)
  from *this* **show** *False* **by** *blast*
qed

Pored toga postoji i narednih nekoliko naredbi koje skraćuju dužinu dokaza (bez ugrožavanja čitljivosti samog dokaza) ili reorganizuju činjenice korišćene u dokazu (na primer, *using* naredba pomera činjenice nakon tvrđenja koje se trenutno dokazuje):

then	=	from *this*
thus	=	then show
hence	=	then have
with *facts*	=	from *facts this*
have *prop* using *facts*	=	from *facts* have *prop*

Dodatno, naredba *using* se može koristiti za uključivanje pomoćne (prethodno dokazane) leme u tekući dokaz.

Uz pomoć ovih skraćenica, prethodni dokaz možemo zapisati i na sledeći način:

proof
  assume *surj f*
  hence $\exists\, a.\ \{x.\ x \notin f\ x\} = f\ a$ **by**(*auto simp*: *surj-def*)
  thus *False* **by** *blast*
qed

4.2.2 Struktuirani ispis tvrđenja

Još jedan način na koji možemo tvrđenja koja dokazujemo učiniti čitljivijim, jeste korišćenje struktuiranog ispisa samog tvrđenja uz pomoć naredbi *fixes*, *assumes*, *shows*.

- Opciona naredba *fixes* se koristi za definisanje tipa promenljivih. Povećava čitljivost našeg koda, jer se tip promenljive izmešta iz samog tvrđenja i

pomera ispred njega.

- Naredba *assumes* se koristi za izdvajanje pretpostavki tvrđenja koje se dokazuje. U slučaju da tvrđenje ima više pretpostavki, one mogu biti razdvojene veznikom *and* i mogu biti imenovane po potrebi uvođenjem labela (navođenjem imena koga prati dvotačka pre same formule *name*:). U ovom primeru je navedeno ime *s* za pretpostavku *surj f*.

- Naredba *shows* zadaje cilj tvrđenja. U ovom konkretnom slučaju tvrđenje koje dokazujemo je $surj\ f \Longrightarrow False$.

- Zapis leme u obliku *assumes* − *shows* implicitno uvodi promenljivu sa imenom *assms* koja predstavlja kompletnu listu svih pretpostavki tvrđenja koje se dokazuje. Pojedinačne pretpostavke su označene svojim rednim brojevima *assms*(*1*), *assms*(*2*), itd. One se mogu koristiti ili u okviru samog dokaza po potrebi, ili se mogu uključiti u sam dokaz na samom početku naredbom *using assms* (pre poziva naredbe *proof*).

Sada, Kantorovu teoremu možemo zapisati na naredni način:

lemma
  fixes $f :: 'a \Rightarrow 'a\ set$
  assumes *s*: *surj f*
  shows *False*

Dodatno, nekada se može desiti da je u okviru *from* naredbe potrebno koristiti neku od ranije poznatih (ili dokazanih) činjenica. Njih možemo koristiti na dva načina. Ili direktno, referisanjem upravo na njihovu vrednost kada ih navodimo između obrnutih jednostrukih navodnika ‘ i ‘ ili između ⟨⟩ uglastih zagrada.[3]

have $x > 0$
⋮
from *‘x>0‘* ...

Ili možemo činjenici dodeliti labelu i možemo referisati na tu činjenicu preko imena te labele. Ovakve činjenice se mogu koristiti i u okviru naredbe *from* i nakon naredbe *using*.

Ovako formulisano tvrđenje može imati naredni dokaz.

proof −
  have $\exists\ a.\ \{x.\ x \notin f\ x\} = f\ a$ **using** *s*

[3]Uglaste zagrade se ispisuju u Isabelle/HOL ili kao `\<open>\<close>` ili kada se otkuca dvostruki navodnik " i sačeka se par sekundi da Isabelle/HOL izgeneriše par otvorenih-zatvorenih uglastih zagrada koje se dobijaju pritiskom na dugme Tab.

by(*auto simp*: *surj-def*)
thus *False* **by** *blast*
qed

U nekim situcijama neće nam odgovarati da Isabelle/HOL automatski izabere pravilo prirodne dedukcije kojim će transformisati tekući cilj (što je podrazumevano ponašanje naredbe *proof*) i biće potrebno da tu automatizaciju zaustavimo. Način na koji se to radi je navođenjem crtice nakon ključne reči: *proof* −.

Napomenimo da je struktura poslednjeg dokaza takva da nam upravo odgovara da pozovemo naredbu *proof* − (sa crticom). Korišćenje crtice nam omogućava da pokrenemo praznu komandu koja započinje proces dokazivanja cilja ali bez ikakve transformacije cilja (pogledajte šta bi se desilo u slučaju da se ona izostavi). To da li će naredba *proof* biti korišćena sa ili bez crtice zavisiće od konkretnog slučaja.

4.3 Najčešće korišćeni logički šabloni u dokazima

Prilikom pisanja Isar dokaza, u situaciji kada pozivamo naredbu *proof* bez ikakvih argumenata, u pozadini se automatski pozivaju pravila prirodne dedukcije. Njih smo već smo detaljno obradili u poglavlju 3[4], a u ovom poglavlju ćemo pokazati na koji način se grade Isar dokazi koji se oslanjaju na ta pravila [5]. Pored pravila prirodne dedukcije koja se implicitno pozivaju (i odnose se na standardne veznike matematičkih formula - negacija, disjunkcija, ekvivalencija itd.), prikazaćemo na koji način i uz pomoć kojih metoda se grade dokazi grananjem i dokazi kontradikcijom.

Naredba *proof* transformiše zaključak tekućeg tvrđenja. U narednim primerima, svi šabloni će biti navedeni u kombinaciji sa *show* naredbom, ali se isti okakvi šabloni mogu upotrebiti i u kombinaciji sa naredbama *have* i *lemma* što će biti prikazano u kasnijim primerima.

4.3.1 Dokazivanje ekvivalencije

Kada je cilj koji se dokazuje zapisan u obliku ekvivalencije, primena naredbe *proof* automatski generiše, kao što je i očekivano, grananje u kome dokazujemo prvo jedan smer ekvivalencije, pa nakon toga drugi smer ekvivalencije:

[4] dodaj referencu
[5] http://mirror.clarkson.edu/isabelle/dist/library/Doc/Prog_Prove/index.html

1. $P \Longrightarrow Q$
2. $Q \Longrightarrow P$

Dve grane dokaza se razdvajaju ključnom rečju *next*, a u svakoj grani dokaza implikacija koja se dokazuje se zapisuje u okviru ključnih reči *assume* (nakon koje se navodi pretpostavka) i *show* (nakon koje se navodi zaključak).

Dokaz novog tvrđenja se može zapisati ili izvođenjem unapred (u prostoru označenim vertikalnim tačkicama) ili izvođenjem unazad (u prostoru označenim rečju *proof* u uglastim zagradama).

show $P \longleftrightarrow Q$
proof
  assume P
  $\vdots$
 show Q $\langle proof \rangle$
next
  assume Q
  $\vdots$
 show P $\langle proof \rangle$
qed

4.3.2 Analiza slučaja

U ovom primeru cilj koji se dokazuje je označen sa R i biće dokazan kroz analizu slučaja u klasičnoj logici. Dokazali smo da u klasičnoj logici važi tvrđenje $P \vee \neg P$ pa će se dokaz granati na dve grane dokaza u zavisnosti od toga da li važi P ili $\neg P$. U suštini primenjuje se pravilo: $(P \Longrightarrow R) \Longrightarrow (\neg P \Longrightarrow R) \Longrightarrow R$.

Navođenjem ključne reči *cases* nakon naredbe *proof*, Isabelle/HOL prepoznaje da treba da formira dva cilja u zavisnosti od (u tom trenutku nepoznate) formule P. Konkretna formula P se navodi nakon ključne reči *assume*. Formula P će biti na odgovarajući način odabrana (u zavisnosti od samog tvrđenja teoreme).

Drugi šablon prikazuje kako se metod *cases* može odmah instancirati sa konkretnom formulom. U tom slučaju su dve grane označene slučajevima *case True*, odnosno *case False*.

show R
proof *cases*
 assume P
 ⋮
 show R ⟨*proof*⟩
next
 assume $\neg\ P$
 ⋮
 show R ⟨*proof*⟩
qed

lemma $\neg\ A \vee A$
proof (*cases* A)
 case *True*
 ⋮
 show *?thesis* ⟨*proof*⟩
next
 case *False*
 ⋮
 show *?thesis* ⟨*proof*⟩
qed

4.3.3 Eliminacija disjunkcije

Eliminacija disjunkcije se odnosi na situaciju kada znamo da važi $P \vee Q$, i iz tog tvrđenja direktno želimo da izvedemo cilj R. Zbog toga ovaj šablon ima ključnu reč *then* (za koju smo već rekli da je skraćeni zapis kombinacije ključnih reči *from this*) ispred ključne reči *show*. Kada imamo ovakvu situaciju, dokaz nastavljamo u dve grane u zavisnosti od toga da li važi P ili Q.

have $P \vee Q$ ⟨*proof*⟩
then show R
proof
 assume P
 ⋮
 show R ⟨*proof*⟩
next
 assume Q
 ⋮
 show R ⟨*proof*⟩
qed

4.3.4 Dokazivanje negacije i dokaz kontradikcijom

Kada je primarni veznik u formuli koja se dokazuje negacija, koristi se šablon sa leve strane. Pravilo koje se implicitno primenjuje je uvođenje negacije i generiše cilj $R \Longrightarrow$ *False*.

Sa desne strane, možemo da vidimo kako se pravilo klasične kontradikcije *ccontr* direktno poziva u Isar dokazima nakon čega cilj postaje $\neg\ R \Longrightarrow$ *False*. Obe implikacije se dokazuju na standardan način, uz pomoć ključnih reči *assume* i *show*.

show $\neg\ R$
proof
  assume R
  ⋮
 show *False* ⟨*proof*⟩
qed

show R
proof (*rule ccontr*)
  assume $\neg R$
  ⋮
 show *False* ⟨*proof*⟩
qed

4.3.5 Univerzalni i egzistencijalni kvantifikator

Sa leve strane je prikazan šablon za dokazivanje cilja koji počinje univerzalnim kvantifikatorom. Implicitno se primenjuje pravilo uvođenja univerzalnog kvantifikatora i cilj se transformiše u $\bigwedge x.\ R\ x$.

Nakon toga, dokaz počinje ključnom rečju *fix*, nakon koje se navodi promenljiva x koja u lokalnom poddokazu ima fiksiranu vrednost. Za tu promenljivu x treba pokazati da važi tvrđenje $R(x)$. Njeno značenje odgovara matematičkom *proizvoljna ali fiksirana vrednost* i nije ograničeno na promenljivu x već može imati proizvoljan naziv.

Sa desne strane je prikazan šablon za dokazivanje cilja koji počinje egzistencijalnim kvantifikatorom. U dokazu koji sledi, potrebno je identifikovati svedoka. Term *svedok* označava proizvoljan term koji zadovoljava svojstvo R.

show $\forall\, x.\ R(x)$
proof
  fix x
  ⋮
 show $R(x)$ ⟨*proof*⟩
qed

show $\exists\, x.\ R(x)$
proof
  ⋮
 show $R(svedok)$ ⟨*proof*⟩
qed

4.3.6 Eliminacija egzistencijalnog kvantifikatora

Ako je potrebno da eliminišemo egzistencijalni kvantifikator iz neke formule, možemo koristiti naredni šablon:

have $\exists\, x.\ P(x)$ ⟨*proof*⟩
then obtain x **where** p: $P(x)$ **by** *blast*

Promenljiva x koja se navodi nakon ključne reči *obtain* je fiksirana lokalna promenljiva. Ime promenljive je proizvoljno. U dokazu je korišćeno i imenovanje uvedene činjenice $P(x)$, nakon ove naredbe na tu činjenicu možemo referisati preko labele p. Ovakav šablon se primenjuje i u situaciji kada uvodimo više promenljivih.

Primer korišćenja naredbe *obtain* videćemo u narednoj verziji dokaza Kantorove teoreme:

lemma $\neg$ *surj*$(f :: 'a \Rightarrow 'a\ set)$
proof
  assume *surj f*
  hence $\exists a.\ \{x.\ x \notin f\ x\} = f\ a$ **by**(*auto simp*: *surj-def*)
  then obtain *a* **where** $\{x.\ x \notin f\ x\} = f\ a$ **by** *blast*
  hence $a \notin f\ a \longleftrightarrow a \in f\ a$ **by** *blast*
  thus *False* **by** *blast*
qed

4.3.7 Relacije jednakost i podskup nad skupovima

Relacija jednakosti skupova se dokazuje, isto kao u standardnoj matematici, u dva smera i prikazana je na levoj strani. Nakon toga, svaki smer se može dokazati korišćenjem šablona prikazanog na desnoj strani.

U desnom delu prikazan je šablon dokazivanja relacije podskup nad dva skupa. Nakon naredbe *proof* cilj se implicitno transformiše u $\bigwedge x.\ x \in A \Longrightarrow x \in B$. Ovo je univerzalno kvantifikovana formula koja se dokazuje prema šablonu koji je već opisan. Uvodi se fiksirana ali proizvoljna promenljiva x, uvodimo pretpostavku da x pripada skupu A i dokazuje se da x pripada i skupu B.

show $A = B$
proof
  show $A \subseteq B$ $\langle proof \rangle$
next
  show $B \subseteq A$ $\langle proof \rangle$
qed

show $A \subseteq B$
proof
  fix x
  assume $x \in A$
  $\vdots$
  show $x \in B$ $\langle proof \rangle$
qed

4.4 Skraćeni zapis dokaza

U ovom poglavlju biće prikazano nekoliko načina na koji se Isar dokazi mogu zapisati na čitljiviji i često kraći način nego što je do sada bio slučaj.

4.4.1 Nizovi jednakosti

Niz izvođenja jednakosti u matematičkim udžbenicima se često zapisuje na sledeći način:

$$\begin{aligned} t_1 &= t_2 && \langle \textit{justification} \rangle \\ &= t_3 && \langle \textit{justification} \rangle \\ &\vdots \\ &= t_n && \langle \textit{justification} \rangle \end{aligned}$$

U Isar-u ovaj dokaz bi bio prikazan na sledeći način:

have $t_1 = t_2$ $\langle proof \rangle$
also have ... $= t_3$ $\langle proof \rangle$
$\vdots$
also have ... $= t_n$ $\langle proof \rangle$
finally show $t_1 = t_n$.

Promenljiva koja omogućava ovakvo izvršavanje dokaza se naziva *calculation* (nalik promenljivoj *this*). Nakon prvog poziva naredbe *also*, Isabelle/HOL postavlja vrednost promenljive *calculation* na *this*. Nakon svakog narednog poziva naredbe *also*, Isabelle/HOL kombinuje vrednosti ove dve promenljive koristeći pravila tranzitivnosti nad relacijama $=$, $\leq$ i $<$ i izračunava novu vrednost promenljive *calculation*.

U ovom dokazu se koriste naredne oznake:

"..." oznaka predstavlja doslovno tri tačke jedna do druge. One zajedno predstavljaju vrednost promenljive koja se javlja sa desne strane jednakosti u prethodnoj Isar naredbi.

"finally" ključna reč *finally*, skuplja do tada izvedene jednakosti u formulu oblika $t_1 = t_n$.

"." oznaka predstavlja jednu tačku i označava metod dokazivanja koji smo ranije videli kroz naredbu *assumption*, odnosno dokazuje cilj kada se on nalazi među dobijenim pretpostavkama.

Naredba **show** $t_1 = t_n$ eksplicitno navodi cilj koji se dokazuje. Kako je taj cilj isti kao formula izvedena ključnom rečju *finally*, metod "." se upotrebljava i dokazuje cilj na osnovu pretpostavke.

Ovakav tip dokaza se može koristiti sa proizvoljnom kombinacijom operatora $=$, $\leq$ i $<$, na primer:

have $t_1 < t_2$ $\langle proof \rangle$
also have ... $= t_3$ $\langle proof \rangle$

⋮
also have ... $\leq t_n$ $\langle proof \rangle$
finally show $t_1 < t_n$.

Simbol relacije koji se koristi u poslednjem **finally** koraku mora biti najopštiji mogući. U prethodnom primeru to znači da ne smemo da koristimo $t_1 \leq t_n$ već moramo da zapišemo $t_1 < t_n$.

Napomena: Isabelle/HOL podržava relacije $=$, $\leq$ i $<$, ali ne podržava relacije $\geq$ i $>$. Otuda ako u dokazu ima potrebe za ovim relacijama, izvođenje se mora izvršiti u obrnutom smeru.

4.4.2 Definisanje skraćenica umesto formula

Vrlo često se formule koje se javljaju u dokazu ponavljaju na više mesta. U slučaju jednostavnijih tvrđenja ovo ne mora predstavljati problem, međutim kada se dokazuju duže formule kao i one čiji je zapis komplikovaniji ovo može predstavljati dodatni napor koji se može izbeći.

Definisanje ključa

Uvođenjem skraćenica može se izbeći ponovno kucanje dugačkih formula. Definišu se prilikom navođenja same formule (nakon ključne reči *show*, *have* ili *lemma*), tako što se unutar običnih zagrada nakon ključne reči *is* navede ključ po kome se vrši uparivanje promenljivih. Promenljive se navode nakon znaka pitanja: *show* \textit{*formula*} ({*is*} \textit{*ključ*}).

U slučaju formule koja dokazuje ekvivalenciju oblika $formula_1 \longleftrightarrow formula_2$ možemo uvesti ključ naredbom (*is* *?L* $\longleftrightarrow$ *?R*). Dokaz se dalje kreira na način koji je već ranije objašnjen prilikom uvođenja šablona za dokazivanje ekvivalencije. U dokazu će biti korišćene samo skraćenice *?L* i *?R* (umesto originalnih potencijalno dugačkih formula).

show $formula_1 \longleftrightarrow formula_2$ (**is** *?L* $\longleftrightarrow$ *?R*)
proof
 assume *?L*
 ⋮
 show *?R* $\langle proof \rangle$
next
 assume *?R*
 ⋮
 show *?L* $\langle proof \rangle$
qed

Skraćenica *thesis*

Promenljiva sa imenom *?thesis* se koristi umesto cilja u tekućem dokazu koji je naveden nakon *lemma* ili *show* naredbe. Ovakvo instanciranje se vrši implicitno. Na primer:

lemma *formula*
proof −
⋮
show *?thesis* ⟨*proof*⟩
qed

Instanciranje promenljivih

Nove promenljive mogu biti uvedene nakon ključne reči *let* i mogu biti korišćenje u narednim koracima dokaza:

let *?t* = *some-big-term*

have ... *?t* ...

Napomena: za razliku od labela koje se uvode za već izvedene činjenice (naredbom oblika *name*:), promenljive ovog oblika *?x* se odnose na terme ili formule.

4.4.3 *moreover* — *ultimatelly* struktuiranje dokaza

U situaciji kada finalno tvrđenje zavisi od niza ranije izvedenih činjenica možemo očekivati format dokaza prikazan sa leve strane. U njemu su uvedene oznake lab_1, lab_2, ..., lab_n, koje su korišćene u poslednjem koraku dokaza neposredno pre izvođenja željene činjenice P.

Drugi način za zapisivanje ovakvog dokaza je prikazan šablonom sa desne strane. Korišćenje ključne reči *moreover*, nakon dokaza svakog dodatnog tvrđenja, i na kraju korišćenja ključne reči *ultimatelly* koja zamenjuje red *from* lab_1 lab_2 ... sa leve strane. Na ovaj način se dobija struktuiran dokaz u kome nema potrebe za uvođenjem novih imena.

have lab_1: P_1 ⟨*proof*⟩
have lab_2: P_2 ⟨*proof*⟩
⋮
have lab_n: P_n ⟨*proof*⟩
from lab_1 lab_2 ...
have P ⟨*proof*⟩

have P_1 ⟨*proof*⟩
moreover have P_2 ⟨*proof*⟩
moreover
⋮
moreover have P_n ⟨*proof*⟩
ultimately have P ⟨*proof*⟩

5

Dokazi sa osnovnim matematičkim pojmovima

5.1 Algebra skupova

Kao što smo navikli u matematici, skupove ćemo označavati velikim latiničnim slovima: A, B, C,... dok ćemo elemente koji sačinjavaju skup označavati malim latiničnim slovima: a, b, c,... Skup elemenata tipa *'a* se označava *'a set*.

Isto kao i u matematici uvodimo pojam univerzalnog skupa koji sadrži sve trenutne elemente, njegova oznaka je *UNIV*. Prazan skup po definiciji ne sadrži ni jedan element i označava se samo sa otvorenom i zatvorenom vitičastom zagradom {}, skup koji sadrži konačan skup elemenata označava se $\{e_1,\ldots,e_n\}$.

Operacije nad skupovima zapisujemo na uobičajeni način:

- Informaciju da element a pripada skupu A, odnosno $a \in A$ zapisujemo `a \<in> A`. Drugi način da dobijemo simbol pripadanja skupu jeste da upotrebimo skraćenicu : (dvotačka).

- Informaciju da element a ne pripada skupu A, odnosno $a \in A$ zapisujemo `a \<notin> A`.

- Operacije preseka, unije i razlike zapisujemo uz pomoć standardnih simbola $A \cup B$, $A \cap B$ i $A \setminus B$, odnosno `A \<union> B`, `A \<inter> B` i `A \ B`.

- Komplement skupa se označava crticom $-A$ i označava skup elemenata koji pripadaju univerzalnom skupu ali ne i skupu A.

- Informaciju da je skup A podskup skupa B zapisujemo $A \subset B$ i $A \subseteq B$, odnosnov`A \<subset> B` i `A \<subseteq> B` (u zavisnosti od toga da li je skup pravi podskupvili ne).

- Dva skupa su jednaka ako i samo ako važi $A \subseteq B$ i $B \subseteq A$, i to zapisujemo uobičajenim simbolom jednakosti $A = B$.

- Ako svojstvo P važi za svaki element skupa A to zapisujemo $\forall\, x{\in}A.\ P$, odnosno ako postoji element za koji važi svojstvo P zapisujemo $\exists\, x{\in}A.\ P$.
x

ASCII oblik matematičkih simbola koji se koriste za rad sa skupovima. Levo je prikazan sam simbol, u sredini je prikazan pun naziv, a sa desne strane je prikazan skraćen zapis. Da bi se dobio sam simbol treba sačekati par sekundi da Isabelle/HOL prepozna zapis i da ponudi simbol koji taj zapis generiše.

$\in$	\<in> :
$\notin$	\<notin> ~:
$\subseteq$	\<subseteq> <=
$\cup$	\<union> Un
$\cap$	\<inter> Int

Primeri koje ćemo koristiti u početku će biti jednostavna tvrđenja, od kojih je velika većina već dokazana u Isabelle/HOL. Možemo koristiti naredbu `find_theorems` da prema odgovarajućem šablonu pretražimo bazu znanja već dokazanih teorema. Čak iako to ne uradimo, Isabelle/HOL će prilikom dokazivanja teoreme čiji dokaz već postoji u bazi znanja, teoremu označiti plavim uzvičnikom sa leve strane i ispisati poruku *Auto solve-direct*: *the current goal can be solved directly with...* nakon čega će navesti ime koje ta teorema nosi u Isabelle/HOL.

Neki od narednih primera su preuzeti sa stranice https://en.wikipedia.org/wiki/Algebra_of_sets

find-theorems - $\cup$ (- $\cup$ -)

Prvo možemo istestirati nekoliko uobičajenih teorema nad skupovima:

lemma $A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$
by *auto*

lemma $- (A \cup B) = - A \cap - B$
by *auto*

lemma $A = B \longleftrightarrow A \subseteq B \wedge B \subseteq A$
by *auto*

lemma $A = B \longleftrightarrow (\forall\ x.\ x \in A \longleftrightarrow x \in B)$
by *auto*

Naredbu *term* možemo iskoristiti da odredimo tip veznika ili tip nekog skupa podataka. U komentarima će biti ispisan Isabelle/HOL izlaz.

term $(\cap)$ — $'a\ set \Rightarrow 'a\ set \Rightarrow 'a\ set$
term $(\in)$ — $'a \Rightarrow 'a\ set \Rightarrow bool$
term $\{1,2,3\}$ — 'a set
term $\{1::nat,\ 2,\ 3\}$ — nat set

Vidimo da operatori *presek* i *pripadanje* rade nad proizvoljnim tipom podataka ($'a$). Takođe, proizvoljan tip podataka se koristi za tip skupa podataka ($'a\ set$) u slučaju da nije naveden tip elemenata skupa. U slučaju da želimo da naglasimo da je skup elemenata upravo skup prirodnih brojeva,

tip *nat* moramo navesti iza jednog elementa skupa (sa duplom dvotačkom između elementa i njegovog tipa). Dovoljno je navesti tip za jedan element skupa, pošto svi elementi skupa moraju biti istog tipa.

5.1.1 Korak po korak građenje dokaza

U ovom poglavlju ćemo prikazati kako možemo da koristimo Isabelle/HOL za postepeno kreiranje potpuno detaljnog formalnog dokaza u jeziku Isar. Dokaz ćemo kreirati korak po korak i prikazivaćemo sa dosta detalja sve faze dokazivanja. Samim tim imaćemo ponavljanje iste leme nekoliko puta u tekstu zbog prikazivanja različitih koraka dokaza.

Relacija jednakosti skupova i relacija podskup, univerzalno kvantifikovane promenljive

Zadatak 5.1. Napisati detaljan Isar dokaz narednog tvrđenja:
$-\ (A \cup B) = -\ A \cap -\ B$.

Dokaz na prirodnom jeziku

Jednakost skupova znači da treba da dokažemo dva smera, da je levi skup podskup desnog i da je desni skup podskup levog.

Prvi smer: ako pretpostavimo da za proizvoljno ali fiksirano x važi $x \in (A \cup B)^c$, možemo da zaključimo da važi $x \notin A \cup B$. Otuda važi $x \notin A$ i $x \notin B$. To znači da važi $x \in A^c$ i $x \in B^c$, pa možemo da zaključimo da važi i $x \in A^c \cap B^c$.

Drugi smer će biti identičan, samo u drugom smeru.

Detaljno kreiranje Isar dokaza

Svaki Isar dokaz se nalazi između ključnih reči *proof* i *qed*. Ako nakon *proof* stavimo crticu ($-$) onda Isabelle/HOL neće tranformisati cilj koji se dokazuje ni na koji način i sve korake dokaza mora korisnik sam da napiše (što ćemo prikazati malo kasnije). U dokazima iz ovog poglavlja koristićemo *proof* bez crtice. U tom slučaju Isabelle/HOL pokušava da izabere pravilo prirodne dedukcije kojim može da transformiše naš cilj.

U ovom konkretnom slučaju, formula koja se dokazuje je **jednakost skupova** pa Isabelle/HOL automatski primenjuje pravilo uvođenja ekvivalencije i generiše dva odvojena podcilja. Isto kao u standardnoj matematici, da

bismo dokazali jednakost dva skupa dovoljno je i potrebno da dokažemo da važe dva nova podcilja: da je levi skup podskup desnog i da je desni skup podskup levog skupa.

U ovom konkretnom slučaju Isabelle/HOL kreira naredna dva cilja:

1. $-(A \cup B) \subseteq -A \cap -B$

2. $-A \cap -B \subseteq -(A \cup B)$

Na osnovu dva podcilja koje je prepoznao Isabelle/HOL, prvo pišemo kostur dokaza. U situacijama kada imamo više odvojenih ciljeva, svaki od njih moramo nezavisno dokazati. Podcilj navodimo nakon ključne reči *show*. Ako ima više različitih podciljeva, odvajamo ih ključnom rečju *next*. Naredna školjka bi bila prvi korak u pravljenju detaljnog dokaza. U ovom trenutku ćemo dokaze oba podcilja ostaviti za kasnije i zbog toga u startu upotrebljavamo ključnu reč *sorry* (i zbog toga uključujemo *quick-and-dirty* mod).

declare [[*quick-and-dirty* = *true*]]

lemma $-(A \cup B) = -A \cap -B$
proof
  show $-(A \cup B) \subseteq -A \cap -B$
    sorry
next
  show $-A \cap -B \subseteq -(A \cup B)$
    sorry
qed

Sada ćemo pokušati da dokažemo prvi podcilj. I njegov dokaz ponovo počinjemo ključnom rečju *proof* i opet ne stavljamo crticu da bi Isabelle/HOL sam pokušao da transformiše cilj. Trenutni cilj je da pokažemo da je **jedan skup podskup drugog**, tj. da svaki element koji pripada jednom skupu pripada i drugom skupu, odnosno očekujemo da dobijemo **univerzalno kvantifikovan izraz**. To je standardni korak u matematici i upravo ono što generiše Isabelle/HOL:

1. $\bigwedge x.\ x \in -(A \cup B) \Longrightarrow x \in -A \cap -B$

Univerzalno kvantifikovana implikacija se dokazuje uz upotrebu ključnih reči *fix* (kojom uvodimo promenljive), *asumes* (kojom uvodimo pretpostavku) i *show* (kojom uvodimo zaključak).

Sada uz pomoć individualnih *then have* koraka popunjavamo vezu između pretpostavke i traženog zaključka. U ovim dokazima ćemo samo koristiti automatski dokazivač *auto*. Drugi smer se dokazuje na vrlo sličan način. Sam dokaz prati strukturu matematičkog dokaza koji smo već prikazali.

lemma $-(A \cup B) = -A \cap -B$
proof
  show $-(A \cup B) \subseteq -A \cap -B$
  proof
    fix x — fiksiramo univerzalno kvantifikovanu promenljivu
    assume $x \in -(A \cup B)$ — pretpostavka
    then have $x \notin A \cup B$ **by** *auto*
    then have $x \notin A \wedge x \notin B$ **by** *auto*
    then have $x \in -A \wedge x \in -B$ **by** *auto*
    then show $x \in -A \cap -B$ **by** *auto* — zaključak
  qed
next
  show $-A \cap -B \subseteq -(A \cup B)$
  proof
    fix x
    assume $x \in -A \cap -B$
    then have $x \in -A \wedge x \in -B$ **by** *auto*
    then have $x \notin A \wedge x \notin B$ **by** *auto*
    then have $x \notin (A \cup B)$ **by** *auto*
    then show $x \in -(A \cup B)$ **by** *auto*
  qed
qed

Logička ekvivalencija, implikacija i negacija

Zadatak 5.2. Napisati detaljan Isar dokaz narednog tvrđenja:
$\neg (A \vee B) \longleftrightarrow \neg A \wedge \neg B$.

U ovom tvrđenju glavni veznik nam je **ekvivalencija**, pa stoga očekujemo da prvi korak dokaza bude uvođenje ekvivalencije. Kada pustimo Isabelle/HOL da sam izabere prvi korak dokaza vidimo da se upravo to i dešava i u ovom konkretnom slučaju dobijamo naredna dva podcilja:

1. $\neg (A \vee B) \Longrightarrow \neg A \wedge \neg B$
2. $\neg A \wedge \neg B \Longrightarrow \neg (A \vee B)$

Sada dokazujemo prvu implikaciju. Ciljevi koji imaju kao glavni veznik **implikaciju** se dokazuju uz pomoć ključnih reči *assume* i *show*. Sada možemo da zapišemo prvu školjku dokaza.

Da ne bismo svaki put prepisivali podformule koje se javljaju u tvrđenju možemo koristiti **definisanje ključa** nakon navođenja tvrđenja koje želimo da dokažemo (u zagradi se navode ključna reč *is* nakon koje navodimo izraz koji očekujemo, ispred promenljivih koje treba da instanciramo navodimo znak pitanja).

lemma $\neg\ (A \vee B) \longleftrightarrow \neg\ A \wedge \neg\ B$ (**is** *?levo* $\longleftrightarrow$ *?desno*)
proof
  assume *?levo*
  show *?desno*
    sorry
next
  assume *?desno*
  show *?levo*
    sorry
qed

Naredni korak dokaza je raspisivanje prvog podcilja koji glasi $\neg\ (A \vee B) \Longrightarrow \neg\ A \wedge \neg\ B$. Kako je zaključak konjunkcija, znamo da je potreban i dovoljan uslov da ona bude dokazana to da su nezavisno dokazana oba konjukta. Upravo to se i dešava kada pustimo Isabelle/HOL da sam pokuša da transformiše tekući cilj, i dobijamo naredna dva cilja:

1. $\neg\ A$
2. $\neg\ B$

lemma $\neg\ (A \vee B) \longleftrightarrow \neg\ A \wedge \neg\ B$ (**is** *?levo* $\longleftrightarrow$ *?desno*)
proof
  assume *?levo*
  show *?desno*
  proof
    show $\neg\ A$
      sorry
  next
    show $\neg\ B$
      sorry
  qed
next
  assume *?desno*
  show *?levo*
    sorry
qed

U sledećem koraku, tekući cilj $\neg\ A$ u sebi ima **negaciju** kao vodeći veznik. Ponovo ćemo pustiti Isabelle/HOL sam da transformiše ovaj cilj i vidimo da je identifikovao novi cilj $A \Longrightarrow$ *False*, što odgovara pravilima prirodne dedukcije.

1. $A \Longrightarrow$ *False*

Kao što smo već rekli implikacije se dokazuju uz pomoć ključnih reči *assume* i *show*. Na sličan način ćemo dokazati i drugi podcilj, odnosno $\neg\ B$.

1. $B \Longrightarrow False$

lemma $\neg\ (A \vee B) \longleftrightarrow \neg\ A \wedge \neg\ B$ (**is** *?levo* $\longleftrightarrow$ *?desno*)
proof
  assume *?levo*
  show *?desno*
  proof
    show $\neg\ A$
    proof

— naredni cilj je $A \Longrightarrow False$. Odnosno cilj koji se dokazuje sa assume "A" ... show False blokom.

      assume A
      then have $A \vee B$ **by** *auto*
      from *this* **and** ‹*?levo*›
      show *False* **by** *auto*
    qed
  next
    show $\neg\ B$
    proof
      assume B
      then have $A \vee B$ **by** *auto*
      from *this* **and** ‹*?levo*›
      show *False* **by** *auto*
    qed
  qed
next
  assume *?desno*
  show *?levo*
    sorry
qed

Nakon ovoga ostaje nam da dokažemo drugi (glavni) podcilj $\neg\ A \wedge \neg\ B \Longrightarrow \neg\ (A \vee B)$. U ovoj formuli pretpostavka je konjunkcija pa se nje prvo oslobađamo (naredbom *from this have* ‹$\neg\ A$› ‹$\neg\ B$› *by auto*). U zaključku je glavni veznik negacija pa primenjujemo isti šablon kao u prethodnom slučaju i dokazujemo $A \vee B \Longrightarrow False$. Kako se sada **u pretpostavkama nalazi disjunkcija**, prema **šablonu za analizu slučajeva** dokazujemo da tekući cilj (odnosno kontradikcija) sledi nezavisno pod pretpostavkom da važi prvi odnosno drugi disjunkt.

lemma $\neg\ (A \vee B) \longleftrightarrow \neg\ A \wedge \neg\ B$ (**is** *?levo* $\longleftrightarrow$ *?desno*)
proof
  assume *?levo*
  show *?desno*
  proof

show ¬ *A*
proof
— naredni cilj je $A \Longrightarrow$ *False*. Odnosno cilj koji se dokazuje sa assume "A" ... show False blokom.
assume *A*
then have $A \vee B$ **by** *auto*
from *this* **and** ⟨*?levo*⟩
show *False* **by** *auto*
qed
next
show ¬ *B*
proof
assume *B*
then have $A \vee B$ **by** *auto*
from *this* **and** ⟨*?levo*⟩
show *False* **by** *auto*
qed
qed
next
assume *?desno*
— kako u pretpostavkama imamo konjunkciju, možemo je odmah podeliti na dva konjunkta i osloboditi se veznika
from *this* **have** ¬ *A* ¬ *B* **by** *auto*
show *?levo*
proof
— slično kao ranije, Isabelle/HOL tekući cilj transformiše u izraz oblika $P \Longrightarrow$ *False* koji se dokazuje assume "P" ... show False blokom
assume $A \vee B$
from *this* **show** *False*
— ovaj put ćemo prvo zapisati cilj (uz pomoć *show*) pa onda dokaz. U ovakvim dokazima potrebno je dodati naredbu *from this*, da bi pretpostavka bila uključena
proof
— kada imamo dva cilja $A \Longrightarrow$ *False* i $B \Longrightarrow$ *False* oba dokazujemo sa *assume* − *show False* blokom, razdvojena sa *next*
assume *A*
from *this* **and** ⟨¬ *A*⟩ **show** *False* **by** *auto*
next
assume *B*
from *this* **and** ⟨¬ *B*⟩ **show** *False* **by** *auto*
qed
qed
qed

5.1.2 Dodatni primeri

Zadatak 5.3. Dokazati da važi: $A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$

lemma
$A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$ (**is** *?levo = ?desno*)
proof
show *?levo* $\subseteq$ *?desno*
proof
— univerzalni kvantifikator: fix - assume - show blok
fix x
assume $x \in$ *?levo*
then have $x \in A$ $x \in B \cup C$ **by** *auto*
then have $x \in B \vee x \in C$ **by** *auto*
then show $x \in$ *?desno*
proof
— disjunkcija u pretpostavkama se eliminiše i dobijaju se naredna dva cilja (slučaj kada element pripada skupu B i slučaj kada element pripada skupu C) koji imaju implikaciju u sebi a nju dokazujemo blokom *assume − show*; ako ih ima više, razdvajamo ih sa *next*
assume $x \in B$
from *this* **and** ‹$x \in A$› **have** $x \in A \cap B$ **by** *auto*
from *this* **show** $x \in$ *?desno* **by** *auto*
next
assume $x \in C$
from *this* **and** ‹$x \in A$› **have** $x \in A \cap C$ **by** *auto*
from *this* **show** $x \in$ *?desno* **by** *auto*
qed
qed
next
show *?desno* $\subseteq$ *?levo*
proof
— univerzalni kvantifikator, koristimo fix - assume - show blok
fix x
assume $x \in$ *?desno*
then have $x \in A \cap B \vee x \in A \cap C$ **by** *auto*
then show $x \in$ *?levo*
proof
— disjunkciju u pretpostavkama eliminišemo i dobijamo dva slučaja koja raspisujemo dalje sa assume - show blokom
assume $x \in A \cap B$
then have $x \in A$ $x \in B$ **by** *auto*
then have $x \in B \cup C$ **by** *auto*
from *this* ‹$x \in A$› **show** $x \in$ *?levo* **by** *auto*

```
    next
      assume x ∈ A ∩ C
      then have x ∈ A x ∈ C by auto
      then have x ∈ B ∪ C by auto
      from this ‹x ∈ A› show x ∈ ?levo by auto
    qed
  qed
qed
```

Zadatak 5.4. Dokazati da važi: $A \cup B = B \cup A$

```
lemma A ∪ B = B ∪ A
proof
  show A ∪ B ⊆ B ∪ A
  proof
    fix x
    assume x ∈ A ∪ B
    from this have x ∈ A ∨ x ∈ B by auto
    from this show x ∈ B ∪ A
    proof
      assume x ∈ A
      from this show x ∈ B ∪ A by auto
    next
      assume x ∈ B
      from this show x ∈ B ∪ A by auto
    qed
  qed
next — simetrično
  show B ∪ A ⊆ A ∪ B
  proof
    fix x
    assume x ∈ B ∪ A
    from this have x ∈ B ∨ x ∈ A by auto
    from this show x ∈ A ∪ B
    proof
      assume x ∈ B
      from this show x ∈ A ∪ B by auto
    next
      assume x ∈ A
      from this show x ∈ A ∪ B by auto
    qed
  qed
qed
```

Zadatak 5.5. Dokazati da važi: $A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$

```
lemma
  A ∪ (B ∩ C) = (A ∪ B) ∩ (A ∪ C) (is ?levo = ?desno)
proof
  show ?levo ⊆ ?desno
  proof
    fix x
    assume x ∈ ?levo
    from this have x ∈ A ∨ x ∈ B ∩ C by auto
— prvo izvedemo disjunkciju pa onda dokaz
    from this show x ∈ ?desno
    proof
      assume x ∈ A
      from this have x ∈ A ∪ B and x ∈ A ∪ C by auto
      from this show x ∈ ?desno by auto
    next
      assume x ∈ B ∩ C
      from this have x ∈ B x ∈ C by auto
      from this have x ∈ A ∪ B x ∈ A ∪ C by auto
      from this show x ∈ ?desno by auto
    qed
  qed
next
  show ?desno ⊆ ?levo
  proof
    fix x
    assume x ∈ ?desno
    from this have x ∈ A ∪ B x ∈ A ∪ C by auto
    have x ∈ A ∨ x ∉ A by auto
— možemo i na ovaj način da napravimo grananje
    from this show x ∈ ?levo
    proof
— sada granamo dokaz po slučajevima
      assume x ∈ A
      from this show x ∈ ?levo by auto
    next
      assume x ∉ A
      from this ‹x ∈ A ∪ B› have x ∈ B by auto
      from ‹x ∉ A› ‹x ∈ A ∪ C› have x ∈ C by auto
      from this ‹x ∈ B› have x ∈ B ∩ C by auto
      from this show x ∈ ?levo by auto
    qed
  qed
```

qed

Zadatak 5.6. Dokazati da važi: $A - (B \cap C) = (A - B) \cup (A - C)$

lemma $A - (B \cap C) = (A - B) \cup (A - C)$
proof
  show $A - (B \cap C) \subseteq (A - B) \cup (A - C)$
  proof
    fix x
    assume $x \in A - (B \cap C)$
    from *this* **have** $x \in A$ $x \notin B \cap C$ **by** *auto*
    from *this* **have** $x \notin B \vee x \notin C$ **by** *auto*
    from *this* **show** $x \in (A - B) \cup (A - C)$
    proof
      assume $x \notin B$
      from *this* ‹$x \in A$› **have** $x \in A - B$ **by** *auto*
      from *this* **show** $x \in (A - B) \cup (A - C)$ **by** *auto*
    next
      assume $x \notin C$
      from *this* ‹$x \in A$› **have** $x \in A - C$ **by** *auto*
      from *this* **show** $x \in (A - B) \cup (A - C)$ **by** *auto*
    qed
  qed
next
  show $(A - B) \cup (A - C) \subseteq A - (B \cap C)$
  proof
    fix x
    assume $x \in (A - B) \cup (A - C)$
    from *this* **have** $x \in A - B \vee x \in A - C$ **by** *auto*
    from *this* **show** $x \in A - (B \cap C)$
    proof
      assume $x \in A - B$
      from *this* **have** $x \in A$ $x \notin B$ **by** *auto*
      from *this* **have** $x \notin B \cap C$ **by** *auto*
      from *this* ‹$x \in A$› **show** $x \in A - (B \cap C)$ **by** *auto*
    next
      assume $x \in A - C$
      from *this* **have** $x \in A$ $x \notin C$ **by** *auto*
      from *this* **have** $x \notin B \cap C$ **by** *auto*
      from *this* ‹$x \in A$› **show** $x \in A - (B \cap C)$ **by** *auto*
    qed
  qed
qed

5.2 Osnovna svojstva funkcija

U ovom poglavlju ćemo dokazati nekoliko lema koje opisuju osnovna svojstva funkcija. Takođe ćemo pokazati nekoliko tvrđenja koja kombinuju skupove i funkcije. Formule nad funkcijama izgledaju isto kao u matematičkim udžbenicima.

Jedna jednostavna lema koja se tiče jednakosti funkcija se može zapisati na sledeći način (i dokazati automatski):

lemma $f = g \longleftrightarrow (\forall\ x.\ f\,x = g\,x)$
by *auto*

Isabelle/HOL već ima definisane funkcije koje proveravaju da li je funkcija injektivna, surjektivna ili bijektivna. Tip sve tri funkcije je $('a \Rightarrow 'b) \Rightarrow$ *bool*, što se može videti pozivanjem sa naredbom *term*:

term *inj*
term *surj*
term *bij*

Ove funkcije su uvedene definicijama, a kao što smo ranije videli, definicije se eksplicitno moraju navesti u dokazu. Definicije su uvedene na uobičajeni način (i njihov naziv odgovara):
inj-def: $inj\ f = (\forall\, x\ y.\ f\,x = f\,y \longrightarrow x = y)$
surj-def: $surj\ f = (\forall\, y.\ \exists\, x.\ y = f\,x)$
bij-def: $bij\ f = (inj\ f \wedge surj\ f)$.
Naredbom `thm` možemo ispisati te teoreme u Isabelle/HOL:

thm *inj-def*
thm *surj-def*
thm *bij-def*

Postoje dva standardna načina za korišćenje definicija, jedan je korišćenjem naredbe `using`, a drugi je korišćenjem naredbe `unfolding`. Kako naredna lema koristi sve tri definicije, njen dokaz neće proći korišćenjem samo metoda *auto*.
Prvi način:

lemma $bij\ f \longleftrightarrow inj\ f \wedge surj\ f$
— by auto — ne prolazi samo po sebi
using *bij-def* **by** *auto*

Drugi način korišćenjem komande `unfolding` koja se primenjuje direktno na cilj koji dokazujemo primenjujući jednakosti koje su date tom definicijom.

lemma *bij f* $\longleftrightarrow$ *inj f* $\wedge$ *surj f*
unfolding *bij-def*
by *auto*

5.2.1 Slika skupa funkcijom

Neke od osnovnih lema možemo naći na sajtu: https://en.wikipedia.org/wiki/Image_(mathematics)

U tim lemama koristićemo sledeće dve definicije:

- slika funkcije nad skupom elemenata,

image-def: f ‘ $A = \{y.\ \exists x{\in}A.\ y = f\,x\}$,

- inverzna slika funkcije nad skupom elemenata,

vimage-def: f $-$‘ $B \equiv \{x.\ f\,x \in B\}$

thm *image-def*
thm *vimage-def*

lemma f ‘ $(A \cup B) = f$ ‘ $A \cup f$ ‘ B
by *auto*

U narednom dokazu treba voditi računa da, pošto je lema zadata u struktuiranom obliku (*assumes* $-$ *shows*), njene pretpostavke moraju biti eksplicitno uključene u dokaz naredbom `using assms`.

lemma
assumes *inj f*
shows f ‘ $(A \cap B) = f$ ‘ $A \cap f$ ‘ B
— *by auto* nece proći
— *by* (*auto simp add*: *image-Int*) isto ne prolazi zato što dokaz ne vidi pretpostavke pa je neophodno dodati pretpostavke narednom komandom:
using *assms*
by (*auto simp add*: *image-Int*)

Zadatak 5.7. Dokazati da važi naredno tvrđenje:

lemma f ‘ $(A \cup B) = f$ ‘ $A \cup f$ ‘ B
proof
show f ‘ $(A \cup B) \subseteq f$ ‘ $A \cup f$ ‘ B
proof
fix y — Izabrali smo ime y zato što odgovara slici.
assume $y \in f$ ‘ $(A \cup B)$
then have $\exists\ x.\ x \in A \cup B \wedge f\,x = y$ **by** *auto* — Po definiciji f ‘
then obtain x **where** $x \in A \cup B\ f\,x = y$ **by** *auto*

```
    then have x ∈ A ∨ x ∈ B by auto
    then show y ∈ f ‘ A ∪ f ‘ B
    proof
```

— Zbog ključne reči *then* nakon koje sledi disjunkcija, automatski se dobijaju naredna dva podcilja.

```
      assume x ∈ A
      then have f x ∈ f ‘ A by auto
      from this ⟨f x = y⟩ have y ∈ f ‘ A by auto
      from this show y ∈ f ‘ A ∪ f ‘ B by auto
    next
      assume x ∈ B
      then have f x ∈ f ‘ B by auto
      from this ⟨f x = y⟩ have y ∈ f ‘ B by auto
      from this show y ∈ f ‘ A ∪ f ‘ B by auto
    qed
  qed
next
  show f ‘ A ∪ f ‘ B ⊆ f ‘ (A ∪ B)
  proof
    fix y
    assume y ∈ f ‘ A ∪ f ‘ B
```

— Ako skup pripada uniji, onda pripada jednom skupu ili drugom skupu

```
    then have y ∈ f ‘ A ∨ y ∈ f ‘ B by auto
    then show y ∈ f ‘ (A ∪ B)
```

— Na osnovu disjunkcije dobijamo analizu po slučajevima

```
    proof
      assume y ∈ f ‘ A
      then obtain x where x ∈ A f x = y by auto
      from this have x ∈ A ∪ B by auto
      from this ⟨f x = y⟩
      show y ∈ f ‘ (A ∪ B) by auto
    next
      assume y ∈ f ‘ B
      then obtain x where x ∈ B f x = y by auto
      from this have x ∈ A ∪ B by auto
      from this ⟨f x = y⟩
      show y ∈ f ‘ (A ∪ B) by auto
    qed
  qed
qed
```

Zadatak 5.8. Dokazati da za injektivne funkcije važi sledeća formula: $f \text{ ‘ } (A \cap B) = f \text{ ‘ } A \cap f \text{ ‘ } B$

Prikazaćemo dva načina za dokazivanje ove leme, prvo standardno kao što smo do sada videli, pa zatim sa `moreover - ultimately` kombinacijom.

Prvi način:

lemma
  assumes *inj f*
  shows $f\ `\ (A \cap B) = f\ `\ A \cap f\ `\ B$ (**is** *?l* = *?d*)
proof
  show *?l* ⊆ *?d*
  proof
    fix *y*
    assume $y \in$ *?l*
    from *this* **obtain** *x* **where** $x \in A \cap B\ f\ x = y$ **by** *auto*
    from *this* **have** $x \in A\ x \in B$ **by** *auto*
    from *this* ‹*f x* = *y*› **have** $y \in f\ `\ A\ y \in f\ `\ B$ **by** *auto*
    from *this* **show** *y* ∈ *?d* **by** *auto*
  qed
next
  show *?d* ⊆ *?l*
  proof
    fix *y*
    assume *y* ∈ *?d*
    from *this* **have** $y \in f\ `\ A\ y \in f\ `\ B$ **by** *auto*
    from *this* **obtain** *xa* **where** $xa \in A\ f\ xa = y$ **by** *auto*
    from ‹$y \in f\ `\ B$› **obtain** *xb* **where** $xb \in B\ f\ xb = y$ **by** *auto*
    from *this* ‹*f xa* = *y*› **have** *f xa* = *f xb* **by** *auto*
    from *this* ‹*inj f*› **have** *xa* = *xb* **by** (*auto simp add*: *inj-def*)
    from *this* ‹*xa* ∈ *A*› ‹*xb* ∈ *B*› **have** *xa* ∈ *A* ∩ *B* **by** *auto*
    from *this* ‹*f xa* = *y*› **show** *y* ∈ *f* ‘ (*A* ∩ *B*) **by** *auto*
  qed
qed

Drugi način: u delu dokaza može se iskoristiti ključna reč **moreover** kojom se skupljaju činjenice koje se na kraju iskorišćavaju sa **ultimately**.

lemma
  assumes *inj f*
  shows $f\ `\ (A \cap B) = f\ `\ A \cap f\ `\ B$ (**is** *?l* = *?d*)
proof
  show *?l* ⊆ *?d*
  proof
    fix *y*
    assume $y \in f\ `\ (A \cap B)$
    then obtain *x* **where** $x \in A \cap B\ f\ x = y$ **by** *auto*
    then have $x \in A\ x \in B$ **by** *auto*

```
    then have f x ∈ f ‘ A f x ∈ f ‘ B by auto
    from this ⟨f x = y⟩
    show y ∈ f ‘ A ∩ f ‘ B by auto
  qed
next
  show ?d ⊆ ?l
  proof
    fix y
    assume y ∈ f ‘ A ∩ f ‘ B
    then have y ∈ f ‘ A y ∈ f ‘ B by auto

    from ⟨y ∈ f ‘ A⟩
    obtain xa where xa ∈ A f xa = y by auto

    moreover — služi da bi bila jasnija struktura dokaza

    from ⟨y ∈ f ‘ B⟩
    obtain xb where xb ∈ B f xb = y by auto

    ultimately

    have xa = xb using ⟨inj f⟩ by (auto simp add: inj-def)
    from this ⟨xb ∈ B⟩ have xa ∈ B by auto
    from this ⟨xa ∈ A⟩ have xa ∈ A ∩ B by auto
    from this ⟨f xa = y⟩
    show y ∈ f ‘ (A ∩ B) by auto
  qed
qed
```

5.2.2 Inverzna slika skupa

Zadatak 5.9. Dokazati da za surjektivne funkcije važi naredno tvrđenje f ‘ $(f -‘ A) = A$.

Prvo ćemo **uvesti pomoćnu lemu** i naveśćemo joj ime zato što želimo da je iskoristimo u dokazu glavne leme:

```
lemma inv-slika-lemma:
  f ‘ (f -‘ B) ⊆ B
proof
  fix y
  assume y ∈ f ‘ (f -‘ B)
  then obtain x where x ∈ f -‘ B f x = y by auto
```

```
  then have f x ∈ B by auto
  from this ⟨f x = y⟩
  show y ∈ B by auto
qed
```

U prvoj grani narednog dokaza pozivamo se na prethodno dokazanu lemu, naredbom *using*. U drugoj grani dokaza pozivamo *sledgehammer*. Osim toga dokaz je poprilično jednostavan pa ga nećemo detaljnije objašnjavati.

```
lemma assumes surj f
  shows f ‘ (f −‘ A) = A
proof
  show f ‘ (f −‘ A) ⊆ A
    using inv-slika-lemma
    by auto — ovo je već dokazano
next
  show A ⊆ f ‘ (f −‘ A)
  proof
    fix y
    assume y ∈ A
    obtain x where f x = y using ⟨surj f⟩
      — sledgehammer
      by (metis surjD)

    from this ⟨y ∈ A⟩
    have x ∈ f −‘ A by auto

    from this ⟨f x = y⟩
    show y ∈ f ‘ (f −‘ A) by auto
  qed
qed
```

Zadatak 5.10. Dokazati da je kompozicija funkcija injektivna ako su injektivne obe polazne funkcije.

Za kompoziciju funkcija znamo da važi naredno tvrđenje (pod imenom:) *o-apply*: $(f \circ g)\ x = f\ (g\ x)$.

```
thm o-apply
```

```
lemma assumes inj f inj g
  shows inj (f ∘ g)
proof
  fix x y
```

```
  assume (f ∘ g) x = (f ∘ g) y
  then have f (g x) = f (g y) by auto
  then have g x = g y using ⟨inj f⟩ by (auto simp add: inj-def)
  then show x = y using ⟨inj g⟩ by (auto simp add: inj-def)
qed
```

5.2.3 Dodatni primeri

Zadatak 5.11. Dokazati da važi naredno tvrđenje:

```
lemma
  assumes inj f
  shows x ∈ A ⟷ f x ∈ f ‘ A
proof
  assume x ∈ A
  from this
  show f x ∈ f ‘ A  by auto
next
  assume f x ∈ f ‘ A
  then obtain x' where x' ∈ A f x = f x' by auto
  from this ⟨inj f⟩ have x = x' by (auto simp add: inj-def)
  from this ⟨x' ∈ A⟩
  show x ∈ A by auto
qed
```

Zadatak 5.12. Dokazati da važi naredno tvrđenje:

```
lemma f -‘ (f ‘ A) ⊇ A
proof
  fix x
  assume x ∈ A
  from this have f x ∈ f ‘ A by auto
  from this show x ∈ f -‘ (f ‘ A) by auto
qed
```

Zadatak 5.13. Dokazati da važi naredno tvrđenje:

```
lemma
  assumes inj f
  shows f -‘ (f ‘ A) ⊆ A — može i jednako, samo ovo je drugi smer, prvi je
prethodna lema
proof
```

```
  fix x
  assume x ∈ f −‘ (f ‘ A)
  from this have f x ∈ f ‘ A by auto
  then obtain x′ where x′ ∈ A f x = f x′ by auto
  from this assms have x = x′ by (auto simp add: inj-def)
  from this ‹x′ ∈ A› show x ∈ A by auto
qed
```

Zadatak 5.14. Dokazati da važi naredno tvrđenje:

```
lemma
  assumes surj f surj g
  shows surj (f ∘ g)
  unfolding surj-def
proof
  fix z
  obtain y where z = f y using ‹surj f› by auto

  moreover obtain x where y = g x using ‹surj g› by auto

  ultimately have z = f (g x) by auto

  from this show ∃ x. z = (f ∘ g) x by auto
qed
```

Zadatak 5.15. Dokazati da važi naredno tvrđenje:

```
lemma f −‘ (− B) = − (f −‘ B) (is ?l = ?d)
proof
  show ?l ⊆ ?d
  proof
    fix x
    assume x ∈ f −‘ (− B)
    from this have f x ∈ − B by auto
    from this have f x ∉ B by auto
    from this have x ∉ f −‘ B by auto
    from this show x ∈ − (f −‘ B) by auto
  qed
next
  show ?d ⊆ ?l
  proof
    fix x
    assume x ∈ − (f −‘ B)
```

```
    from this have x ∉ f −‘ B by auto
    from this have f x ∉ B by auto
    from this have f x ∈ − B by auto
    from this show x ∈ f −‘ (− B) by auto
  qed
qed

end
```

5.3 Dokazi u matematičkoj logici

```
theory Cas7-vezbe
imports Main

begin
```

U ovom poglalju biće zapisani uz pomoć programskog jezika Isar dokazi nekih lema koje smo već videli u prethodnom poglavlju (kada smo koristili samo *apply* naredbu). Struktura ovih dokaza će biti veoma slična strukturi *apply−skript* dokaza, ali će biti mnogo čitljivija za prosečnog čitaoca i da bi se pratili ovi dokazi nije neophodno koristiti Isabelle/HOL već se dokaz skoro u potpunosti može pratiti iz knjige. Ipak i pored toga, i ovi dokazi će biti iskomentarisani u odgovarajućim situacijama.

Prva razlika između Isar struktuiranih dokaza i nestruktuiranih dokaza, koje smo ranije videli, dolazi iz mogućnosti korišćenja ključnih reči *assumes* i *shows*. U *apply−skript* dokazima smo i samo tvrđenje leme zapisivali isključivo uz pomoć logičkih veznika, i nakon toga smo morali da koristimo (skoro uvek) naredbe za uvođenje implikacije i eliminaciju konjunkcije iz pretpostavki teoreme. U Isar dokazima čija formulacija samog tvrđenja teoreme koristi ključne reči *assumes* i *shows*, ta dva koraka postaju nepotrebna.

5.3.1 Kvantifikatori

U narednim primerima, prvo je navedena originalna formulacija teoreme (koja koristi samo logičke veznike), pa nakon toga je naveden struktuirani zapis tvrđenja. Sve pretpostavke koje se nalaze sa leve strane implikacije se navode nakon ključne reči *assumes*, a cilj se navodi nakon ključne reči *shows*.

U dokazima neće biti direktno korišćena pravila prirodne dedukcije, već će Isabelle/HOL implicitno koristiti odgovarajuća pravila. Samo u prvom

primeru biće eksplicitno navedena pravila koja se koriste ali biće navedena pod komentarima.

Zadatak 5.16. Uz pomoć (implicitnog korišćenja) pravila prirodne dedukcije u Isar-u dokazati da je naredno tvrđenje teorema logike prvog reda:

U narednom dokazu ćemo pretpostavkama i već dokazanim činjenicama pristupati preko njihove vrednosti koju ćemo navoditi između uglastih zagrada ⟨⟩.

declare [[*quick-and-dirty=true*]]

lemma $(\exists\ x.\ P\ x) \wedge (\forall\ x.\ P\ x \longrightarrow Q\ x) \longrightarrow (\exists\ x.\ Q\ x)$
sorry

lemma
assumes $\exists\ x.\ P\ x$
assumes $\forall\ x.\ P\ x \longrightarrow Q\ x$
shows $\exists\ x.\ Q\ x$
— Pošto je tvrđenje zapisano sa *assumes* i *shows*, neće postojati prva dva koraka iz apply-skript dokaza.
— U slučaju kada je u zaključku egzistencijalni kvantifikator, ne odgovara nam da pustimo Isabelle/HOL sam da izabere pravilo pošto će generisati naredni cilj: *Q ?x*. Tako da u ovom slučaju stavljamo crticu nakon ključne reči *proof* i sami ćemo da ispišemo dokaz.
proof −
— Obratiti pažnju (u Isabelle/HOL - Output prozoru) da je bojama napravljena razlika: zelenom bojom je označeno x u poznatoj činjenici, a narandžastom bojom je označeno x koje je dobijeno eliminacijom egzistencijalnog kvantifikatora.
from ⟨$\exists\ x.\ P\ x$⟩ **obtain** x **where** $P\ x$ **by** *auto*
— U ovim dokazima pravila prirodne dedukcije se koriste implicitno. Ako bismo želeli, mogli bismo da napišemo i eksplicitno, npr. *by* (*rule exE*), ali ovde ne želimo da budemo toliko precizni
from ⟨$\forall\ x.\ P\ x \longrightarrow Q\ x$⟩ **have** $P\ x \longrightarrow Q\ x$ **by** *auto*
— Ili, ako bismo želeli da budemo precizni zapisali bi: *by* (*rule-tac* $x = x$ *in allE*)
from *this* ⟨$P\ x$⟩ **have** $Q\ x$ **by** *auto* — ili *by* (*rule impE*)
from *this* **show** $\exists\ x.\ Q\ x$ **by** *auto* — ili *by* (*rule exI*)
qed

Zadatak 5.17. Ako je svaki covek smrtan; i ako je svaki grk covek; onda važi da je svaki grk smrtan.

Ovo je malo komplikovanija teorema, može se zapisati sa jednom promenljivom x, ali čitljivije je ako uvedemo različite promenljive: c, g i a.

U narednom dokazu ćemo pretpostavkama pristupati preko promenljive *assms*, nakon koje dodajemo redni broj pretpostavke kada želimo da budemo precizni u samom dokazu.

```
lemma (∀ c. covek c ⟶ smrtan c) ∧ (∀ g. grk g ⟶ covek g)
  ⟶ (∀ a. grk a ⟶ smrtan a)
 sorry

lemma
 assumes ∀ c. covek c ⟶ smrtan c
 assumes ∀ g. grk g ⟶ covek g
 shows ∀ a. grk a ⟶ smrtan a
proof
 — Sada nam Isabelle/HOL sugeriše uvođenje fiksiranog ali proizvoljnog a, isto kao da smo napisali proof (rule allI). Uvešćemo Jorgosa na primer.
 fix Jorgos
 show grk Jorgos ⟶ smrtan Jorgos
 proof
  — Opet Isabelle/HOL sugeriše i primenjuje pravilo uvođenja implikacije, isto kao da smo napisali proof (rule impI)
  assume grk Jorgos
  from this and assms(2) have covek Jorgos by auto
  from this and assms(1) show smrtan Jorgos by auto
 qed
qed
```

Ista teorema se može dokazati i na sledeći način korišćenjem *moreover−ultimatelly* konstrukcije.

```
lemma
 assumes ∀ c. covek c ⟶ smrtan c
 assumes ∀ g. grk g ⟶ covek g
 shows ∀ a. grk a ⟶ smrtan a
proof
 fix Jorgos
 show grk Jorgos ⟶ smrtan Jorgos
 proof
  assume grk Jorgos
  moreover
  from assms(2) have grk Jorgos ⟶ covek Jorgos by auto
  ultimately
  have covek Jorgos by auto

— Moguće je koristiti više blokova. Ovde počinje novi blok, prethodno ultimately je ispraznilo sve.
```

```
    moreover
    from assms(1) have covek Jorgos ⟶ smrtan Jorgos by auto
    ultimately
    show smrtan Jorgos by auto
  qed
qed
```

Dodatni primeri

Zadatak 5.18. $(\forall\ a.\ P\ a \longrightarrow Q\ a) \wedge (\forall\ b.\ P\ b) \longrightarrow (\forall\ x.\ Q\ x)$

```
lemma (∀ a. P a ⟶ Q a) ∧ (∀ b. P b) ⟶ (∀ x. Q x)
  sorry

lemma
  assumes ∀ a. P a ⟶ Q a
  assumes ∀ b. P b
  shows ∀ x. Q x
proof
  fix x
  show Q x
  proof −
    from assms(2) have P x by auto
    from assms(1) have P x ⟶ Q x by auto
    from this ‹P x› show Q x by auto
  qed
qed
```

Zadatak 5.19. $(\exists\ x.\ A\ x \vee B\ x) \longrightarrow (\exists\ x.\ A\ x) \vee (\exists\ x.\ B\ x)$

```
lemma (∃ x. A x ∨ B x) ⟶ (∃ x. A x) ∨ (∃ x. B x)
  sorry
```

Ovaj dokaz ćemo pisati u dve faze:

(1) Prvo pravimo samo kostur dokaza, nezavršene dokaze označavamo sa *sorry*.

U ovom dokazu koristimo proof sa crticom (*proof −*), iz razloga što ne želimo da koristimo samo *proof* bez crtice - zato što bi nas primena takvog metoda na zaključak u kome se nalazi disjunkcija dovela do toga da se cilj transformiše tako da dokazujemo samo prvi disjunkt, a on ne mora da važi uvek.

Napomena: pogledajte u Isabelle/HOL šta se dešava kada pokrenete bez crtice.

lemma
assumes $\exists\ x.\ A\ x \vee B\ x$
shows $(\exists\ x.\ A\ x) \vee (\exists\ x.\ B\ x)$
proof −
from *assms* **obtain** x **where** $A\ x \vee B\ x$ **by** *auto*
from *this* **show** $(\exists\ x.\ A\ x) \vee (\exists\ x.\ B\ x)$
proof
— Sada zbog naredbe "from this" koja se odnosi na disjunkciju, dobijamo rezonovanje po slučajevima:
assume $A\ x$
show $(\exists\ x.\ A\ x) \vee (\exists\ x.\ B\ x)$
sorry
next
assume $B\ x$
show $(\exists\ x.\ A\ x) \vee (\exists\ x.\ B\ x)$
sorry
qed
qed

(2) Sada raspisujemo jednu po jednu granu. U prvoj grani ćemo zapisati najkraći dokaz, a drugu granu ćemo malo raspisati.

lemma
assumes $\exists\ x.\ A\ x \vee B\ x$
shows $(\exists\ x.\ A\ x) \vee (\exists\ x.\ B\ x)$
proof −
from *assms* **obtain** x **where** $A\ x \vee B\ x$ **by** *auto*
from *this* **show** $(\exists\ x.\ A\ x) \vee (\exists\ x.\ B\ x)$
proof
assume $A\ x$
from *this* **show** $(\exists\ x.\ A\ x) \vee (\exists\ x.\ B\ x)$ **by** *auto* — ovo bi bio najkraći dokaz
next
assume $B\ x$
show $(\exists\ x.\ A\ x) \vee (\exists\ x.\ B\ x)$
proof −
— U drugoj grani ćemo raspisati dokaz. Obavezno stavljamo - pošto moramo sami da dođemo do drugog disjunkta.
from ‹$B\ x$› **have** $\exists\ x.\ B\ x$ **by** *auto*
from *this* **show** $(\exists\ x.\ A\ x) \vee (\exists\ x.\ B\ x)$ **by** *auto*
qed
qed
qed

Zadatak 5.20. $(\forall\ x.\ A\ x \longrightarrow \neg\ B\ x) \longrightarrow \neg\ (\exists\ x.\ A\ x \wedge B\ x)$

```
lemma (∀ x. A x ⟶ ¬ B x) ⟶ ¬ (∃ x. A x ∧ B x)
  sorry
```

(1) Školjka dokaza

```
lemma
  assumes ∀ x. A x ⟶ ¬ B x
  shows ¬ (∃ x. A x ∧ B x)
proof
  assume ∃ x. A x ∧ B x
  show False
    sorry
qed
```

(2) Raspisani dokaz

```
lemma
  assumes ∀ x. A x ⟶ ¬ B x
  shows ¬ (∃ x. A x ∧ B x)
proof
  assume ∃ x. A x ∧ B x
  from this obtain x where A x ∧ B x by auto
  from this have A x B x by auto
  from this have ¬ B x using assms by auto
  from this ‹B x› show False by auto
qed
```

Narednih nekoliko lema su takve da su njihovi dokazi malo jednostavniji u apply-skript obliku nego kada se koristi Isar.

Zadatak 5.21. Formulisati i dokazati naredno tvrđenje:
Ako za svaki broj koji nije paran važi da je neparan;
i ako za svaki neparan broj važi da nije paran;
pokazati da onda za svaki broj važi da nije istovremeno i paran i neparan.

```
lemma (∀ x. ¬ paran x ⟶ neparan x) ∧ (∀ x. neparan x ⟶ ¬ paran x)
   ⟶ (∀ x. ¬ (paran x ∧ neparan x))
  sorry
```

(1) Kostur Isar dokaza:

```
lemma
  assumes ∀ x. ¬ paran x ⟶ neparan x
  assumes ∀ x. neparan x ⟶ ¬ paran x
  shows ∀ x. ¬ (paran x ∧ neparan x)
proof
```

```
  fix x
  show ¬ (paran x ∧ neparan x)
    sorry
qed
```

(2) Dodajemo naredni korak dokaza:

```
lemma
  assumes ∀ x. ¬ paran x ⟶ neparan x
  assumes ∀ x. neparan x ⟶ ¬ paran x
  shows ∀ x. ¬ (paran x ∧ neparan x)
proof
  fix x
  show ¬ (paran x ∧ neparan x)
  proof
    assume paran x ∧ neparan x
    show False
      sorry
  qed
qed
```

(3) Raspisan dokaz:

```
lemma
  assumes ∀ x. ¬ paran x ⟶ neparan x
  assumes ∀ x. neparan x ⟶ ¬ paran x
  shows ∀ x. ¬ (paran x ∧ neparan x)
proof
  fix x
  show ¬ (paran x ∧ neparan x)
  proof
    assume paran x ∧ neparan x
    from this have paran x neparan x by auto
    from this assms(2) have ¬ paran x by auto
    from this ‹paran x› show False by auto
  qed
qed
```

Zadatak 5.22. Formulisati i dokazati naredno tvrđenje:
Ako za svaki broj koji nije paran važi da je neparan;
i ako za svaki neparan broj važi da nije paran;
pokazati da onda za svaki broj važi da je ili paran ili neparan.

```
lemma (∀ x. ¬ paran x ⟶ neparan x) ∧ (∀ x. neparan x ⟶ ¬ paran x)
```

$\longrightarrow (\forall\ x.\ paran\ x \vee neparan\ x)$
sorry

(1) Prvi korak Isar dokaza:

lemma
assumes $\forall\ x.\ \neg\ paran\ x \longrightarrow neparan\ x$
assumes $\forall\ x.\ neparan\ x \longrightarrow \neg\ paran\ x$
shows $\forall\ x.\ paran\ x \vee neparan\ x$
proof
fix x
show $paran\ x \vee neparan\ x$
sorry
qed

(2) Drugi korak:

lemma
assumes $\forall\ x.\ \neg\ paran\ x \longrightarrow neparan\ x$
assumes $\forall\ x.\ neparan\ x \longrightarrow \neg\ paran\ x$
shows $\forall\ x.\ paran\ x \vee neparan\ x$
proof
fix x
— Dodajemo naredni korak da bi napravili grananje po slučajevima:
have $paran\ x \vee \neg\ paran\ x$ **by** *auto*
from *this* **show** $paran\ x \vee neparan\ x$
— Kada stavimo *from this show* ... , ako je *this* disjunkcija dobijamo automatski dve grane:
proof
assume $paran\ x$
show $paran\ x \vee neparan\ x$
sorry
next
assume $\neg\ paran\ x$
show $paran\ x \vee neparan\ x$
sorry
qed
qed

(3) Treći korak:

lemma
assumes $\forall\ x.\ \neg\ paran\ x \longrightarrow neparan\ x$
assumes $\forall\ x.\ neparan\ x \longrightarrow \neg\ paran\ x$
shows $\forall\ x.\ paran\ x \vee neparan\ x$
proof

```
  fix x
  have paran x ∨ ¬ paran x by auto
  from this show paran x ∨ neparan x
  proof
    assume paran x
    from this show paran x ∨ neparan x by auto
  next
    assume ¬ paran x
    from this have neparan x using assms(1) by auto
    from this show paran x ∨ neparan x by auto
  qed
qed
```

Zadatak 5.23. Ako svaki konj ima potkovice;
i ako ne postoji čovek koji ima potkovice;
i ako znamo da postoji makar jedan čovek;
dokazati da postoji čovek koji nije konj.

```
lemma (∀ x. konj x ⟶ ima-potkovice x) ∧ ¬ (∃ x. covek x ∧ ima-potkovice x)
∧ (∃ x. covek x)
⟶ (∃ x. covek x ∧ ¬ konj x)
  sorry
```

(1) Kostur Isar dokaza:

```
lemma
  assumes ∀ x. konj x ⟶ ima-potkovice x
  assumes ¬ (∃ x. covek x ∧ ima-potkovice x)
  assumes ∃ x. covek x
  shows ∃ x. covek x ∧ ¬ konj x
proof −
  from assms(3) obtain x where covek x by auto
  have konj x ∨ ¬ konj x by auto
  from this have covek x ∧ ¬ konj x
    sorry
  from this show ∃ x. covek x ∧ ¬ konj x by auto
qed
```

(2) Naredni korak raspisan:

```
lemma
  assumes ∀ x. konj x ⟶ ima-potkovice x
  assumes ¬ (∃ x. covek x ∧ ima-potkovice x)
  assumes ∃ x. covek x
```

```
  shows ∃ x. covek x ∧ ¬ konj x
proof −
  from assms(3) obtain x where covek x by auto
  have konj x ∨ ¬ konj x by auto
  from this have covek x ∧ ¬ konj x
  proof
    assume konj x
    show covek x ∧ ¬ konj x
      sorry
  next
    assume ¬ konj x
    show covek x ∧ ¬ konj x
      sorry
  qed
  from this show ∃ x. covek x ∧ ¬ konj x by auto
qed
```

(3) Kompletan dokaz — nije baš elegantan, bolje je koristiti prirodnu dedukciju:

```
lemma
  assumes ∀ x. konj x ⟶ ima-potkovice x
  assumes ¬ (∃ x. covek x ∧ ima-potkovice x)
  assumes ∃ x. covek x
  shows ∃ x. covek x ∧ ¬ konj x
proof −
  from assms(3) obtain x where covek x by auto
  have konj x ∨ ¬ konj x by auto
  from this have covek x ∧ ¬ konj x
  proof
    assume konj x
    show covek x ∧ ¬ konj x
    proof
      from ‹covek x› show covek x by auto
    next
      show ¬ konj x
      proof
        assume konj x
        from this assms(1) have ima-potkovice x by auto
        from ‹covek x› this have ∃ x. covek x ∧ ima-potkovice x by auto
        from this assms(2) show False by auto
      qed
    qed
  next
```

```
    assume ¬ konj x
    from this and ⟨covek x⟩
    show covek x ∧ ¬ konj x by auto
  qed
  from this show ∃ x. covek x ∧ ¬ konj x by auto
qed
```

Zadatak 5.24. Ako je svaki kvadrat romb;
i ako je svaki kvadrat pravougaonik;
i ako znamo da postoji makar jedan kvadrat;
onda postoji makar jedan romb koji je istovremeno i pravougaonik.

```
lemma (∀ x. kvadrat x ⟶ romb x) ∧ (∀ x. kvadrat x ⟶ pravougaonik x) ∧ (∃
x. kvadrat x) ⟶ (∃ x. romb x ∧ pravougaonik x)
  sorry
```

Isar dokaz:

```
lemma
  assumes ∀ x. kvadrat x ⟶ romb x
  assumes ∀ x. kvadrat x ⟶ pravougaonik x
  assumes ∃ x. kvadrat x
  shows ∃ x. romb x ∧ pravougaonik x
proof −
  from assms(3) obtain x where kvadrat x by auto
  from this assms(2) have pravougaonik x by auto
  from ⟨kvadrat x⟩ assms(1) have romb x by auto
  from this ⟨pravougaonik x⟩ show ∃ x. romb x ∧ pravougaonik x by auto
qed
```

Zadatak 5.25. Ako postoji mačka koja nema rep;
i ako je svaka mačka sisar;
onda postoji sisar koji nema rep.

```
lemma (∃ x. macka x ∧ ¬ rep x) ∧ (∀ x. macka x ⟶ sisar x) ⟶
      (∃ x. sisar x ∧ ¬ rep x)
  sorry
```

Isar dokaz:

```
lemma
  assumes ∃ x. macka x ∧ ¬ rep x
  assumes ∀ x. macka x ⟶ sisar x
```

shows $\exists\ x.\ sisar\ x \wedge \neg\ rep\ x$
proof −
from *assms*(*1*) **obtain** x **where** *macka* $x \neg\ rep\ x$ **by** *auto*
from *this assms*(*2*) **have** *sisar* x **by** *auto*
from *this* ‹$\neg\ rep\ x$› **show** $\exists\ x.\ sisar\ x \wedge \neg\ rep\ x$ **by** *auto*
qed

5.3.2 Klasična logika

U ovom poglavlju ćemo prikazati kako se koriste pravila *ccontr* i *classical* u okviru jezika Isar kroz dokaze par lema koje smo već videli u ranijim poglavljima.

Pravilo *ccontr*

Zadatak 5.26. $\neg\ (A \wedge B) \longrightarrow \neg\ A \vee \neg\ B$

Kada je lema zapisana sa logičkim veznicima, bez struktuiranog (*assumes* − *show*) zapisa, prvi korak u dokazu će biti uvođenje implikacije.

U ovom dokazu ćemo prikazati kako možemo da izvedemo među korak u samom dokazu teoreme, nalik pomoćnoj lemi koju ćemo koristiti lokalno u dokazu.

lemma $\neg\ (A \wedge B) \longrightarrow \neg\ A \vee \neg\ B$
proof
— Isabelle/HOL implicitno primenjuje pravilo *impI* (isto kao da smo napisali *proof* (*rule impI*).
assume $\neg\ (A \wedge B)$
show $\neg\ A \vee \neg\ B$ — ovo je naš cilj C
— Napomena: ako ne napišemo ništa nakon *proof*, Isabelle/HOL će ovaj cilj transformisati u prvi disjunkt, što nije korektno, tako da moramo paziti i eksplicitno navesti koje pravilo želimo da bude primenjeno

proof (*rule ccontr*)
— Da bi dokazali cilj C na osnovu pravila *ccontr*, dovoljno je da pokažemo da $\neg\ C \Longrightarrow False$, pa nakon toga sledi *assume* − *show* blok
assume $\neg\ (\neg\ A \vee \neg\ B)$

— Sada ćemo formulisati međukorak koji želimo prvo da dokažemo:
have $A \wedge B$
proof
— Sada nam odgovara da bude samo *proof*, bez crtice, jer Isabelle/HOL sam deli ovaj cilj na dva cilja

```
    show A
— Sada ponovo pozivamo pravilo ccontr.
    proof (rule ccontr)
— Dovoljno je da pokažemo ¬ A ⟹ False odnosno sledi assume − show blok
      assume ¬ A
      then have ¬ A ∨ ¬ B
        by (rule disjI1)
      then show False
        using ‹¬ (¬ A ∨ ¬B)›
— Sve do sada uvedene pretpostavke mogu se koristiti, ali moraju biti navedene svojim vrednostima.
        by auto
    qed
  next
    show B
    proof (rule ccontr)
      assume ¬ B
      then have ¬ A ∨ ¬ B
        by (rule disjI2)
      then show False
        using ‹¬ (¬ A ∨ ¬B)›
        by auto
    qed
  qed
— Sada je dokazano da važi A ∧ B, pa iz toga možemo da izvedemo kontradikciju sa pretpostavkom (uvedenom sa assume).
  then show False
    using ‹¬ (A ∧ B)›
    by auto
qed
qed
```

Zadatak 5.27. Pirsov zakon: $((P \longrightarrow Q) \longrightarrow P) \longrightarrow P$

Ovu lemu ćemo prvo zapisati uz pomoć *assumes* − *shows* struktuiranog zapisa.

```
lemma
  assumes (P ⟶ Q) ⟶ P
  shows P
proof (rule ccontr)
— Dodajemo negaciju zaključka u pretpostavke i pokušavamo da izvedemo kontradikciju.
  assume ¬ P
```

— Dodajemo međukorak:
have $P \longrightarrow Q$
proof
assume P
from *this* ‹$\neg$ P› **show** Q **by** *auto*
qed
— Sada je dokazano da važi $P \longrightarrow Q$ i na to referiše promenljiva *this*.
from *this assms* **have** P **by** *auto*
from *this* ‹$\neg$ P› **show** *False* **by** *auto*
qed

Pravilo *classical*

Zadatak 5.28. $P \vee \neg P$

lemma $P \vee \neg P$
proof (*rule classical*)
— Ovde ne smemo ostaviti bez crtice, zato što Isabelle/HOL transformiše disjunkciju u prvi disjunkt - u ovom trenutku to nam ne odgovara i ne možemo to dokazati zato što još uvek nemamo nikakve pretpostavke; Odmah će biti naglašeno da želimo da koristimo pravilo *classical*.
assume $\neg$ $(P \vee \neg P)$
show $P \vee \neg P$
proof
— Sada je svejedno, jer je simetričan izraz, pa ćemo ostaviti bez crtice i odgovara nam da primeni podrazumevano pravilo *disjI1*.
show P
proof (*rule classical*)
assume $\neg$ P
from *this* **have** $P \vee \neg P$ **by** *auto*
from *this* ‹$\neg$ $(P \vee \neg P)$› **have** *False* **by** *auto*
from *this* **show** P **by** *auto*
qed
qed
qed

Zadatak 5.29. Dokazati klasičan deo de-Morganovog zakona $(\neg\ (\forall\ x.\ P\ x)) \longrightarrow (\exists\ x.\ \neg\ P\ x)$.

Ovu lemu ćemo zapisati uz pomoć *assumes* − *show* struktuiranog zapisa tvrđenja.

lemma *de-Morgan-isar*:
assumes $\neg$ $(\forall\ x.\ P\ x)$

```
  shows ∃ x. ¬ P x
— Ovaj dokaz počinje na isti način kao u apply-skript dokazu.
proof (rule classical)
  assume ¬ (∃ x. ¬ P x) — Imitiramo apply-skript i dodajemo među korak.
  have ∀ x. P x
  proof
— Ostavljamo bez crtice pošto dobijamo oblik koji nam treba: fix − show blok.
    fix x show P x
    proof (rule classical) — isti korak kao u apply-skript dokazu
      assume ¬ P x
      then have ∃ x. ¬ P x by auto
      from this ‹¬ (∃ x. ¬ P x)›
      show P x by auto
    qed
  qed
  from this ‹¬ (∀ x. P x)›
  show ∃ x. ¬ P x by auto
qed
```

5.3.3 Dokazi po slučajevima

Osim linearnih dokaza u kojima se navodi niz linearnih izvođenja, postoje i dokazi koji imaju razgranatu strukturu ili dokazi po slučajevima.

U ovom poglavlju biće prikazano detaljno korišćenje dokazivanja po slučajevima u okviru Isar dokaza.

Paradoks pijanca - The Drinker's principle

Zadatak 5.30. Paradoks pijanca: postoji osoba za koju važi, ako je on pijanac onda su i svi ostali pijanci.

U ovom dokazu se koristi klasični deo de-Morganovog zakona, koji smo dokazali do sada dva puta.

Sada ćemo pogledati dokaz samog paradoksa pijanca u Isar-u. U njemu ćemo koristiti naredbu *proof cases* koja nam omogućava dokazivanje po slučajevima u okviru samog dokaza (kao što smo već pokazali). Dokaz se deli na dva poddokaza, odnosno na dve grane razdvojene sa *next*. Lokalne pretpostavke A (u prvoj grani), odnosno $\neg\ A$ (u drugoj grani), se uvode ključnom rečju *assume*.

Grananje će biti izvršeno prema $\forall\ x.\ drunk\ x$, isto kao u *apply−skript* dokazu.

U narednom dokazu koristi se skraćenica *?thesis* koja se implicitno povezuje sa tekućim ciljem (koji je prethodno bio naveden pod *lemma* ili *show*).

lemma *Drinker's-Principle*: $\exists\ x.\ (drunk\ x \longrightarrow (\forall\ x.\ drunk\ x))$
— Nakon navođenja ključne reči *cases* vidimo da je potrebno identifikovati formulu po kojoj se vrši grananje *?P*
proof *cases*
— Narednom naredbom možemo videti šta je tekući cilj. Skraćenica *?thesis* može biti korišćena nadalje u dokazu umesto eksplicitnog navođenja formule koja se dokazuje (na primer *show ?thesis*).
— Naredna naredba nije deo dokaza, može se obrisati, dodata je zbog ispisa:
term *?thesis* — $\exists x.\ drunk\ x \longrightarrow (\forall x.\ drunk\ x)$
assume $\forall\ x.\ drunk\ x$
— Uvodimo proizvoljno *a*, bez ikakvih dodatnih pretpostavki o njemu.
fix *a*
— Sada bez obzira na to da li važi *drunk a* ili $\neg$ *drunk a*, važiće naredna formula; na osnovu teoreme $A \longrightarrow B \longleftrightarrow \neg\ A \vee B$; odnosno na osnovu leme $B \longrightarrow (A \longrightarrow B)$ (ovde je sa B označeno $\forall\ x.\ drunk\ x$, a sa A je označeno *drunk a*).
from $\langle\forall\ x.\ drunk\ x\rangle$ **have** $drunk\ a \longrightarrow (\forall\ x.\ drunk\ x)$ **by** *auto*
then show *?thesis* **by** *auto*
— Kraj dokaza prve grane.
next
assume $\neg\ (\forall\ x.\ drunk\ x)$
then have $\exists\ x.\ \neg\ drunk\ x$ **by** (*auto simp add*: *de-Morgan-isar*)
then obtain *a* **where** $\neg\ drunk\ a$ **by** *auto*
— Sada pokazujemo da važi naredna implikacija, posto važi $\neg\ A$, važiće i $A \longrightarrow B$ na osnovu leme $\neg\ A \longrightarrow (A \longrightarrow B)$.
have $drunk\ a \longrightarrow (\forall\ x.\ drunk\ x)$
proof
assume *drunk a*
with $\langle\neg\ drunk\ a\rangle$ **show** $\forall\ x.\ drunk\ x$ **by** *auto*
qed
then show *?thesis* **by** *auto*
qed

5.4 Smullyan - Logical Labirinths

Ovo poglavlje biće posvećeno dokazima u iskaznoj, predikatskoj logici koji se nalaze u knjizi: Raymond M. Smullyan, Logical Labyrinths http://logic-books.info/sites/default/files/logical_labirints.pdf

Ovo poglavlje sadrži dosta različitih primera koji mogu poslužiti za vežbanje i primenu metoda koja su već objašnjena u prethodnim poglavljima.

Sada ćemo se podsetiti primera sa početka kursa: Edgar Abercrombie bio je antropolog koga je posebno zanimala logika, sociologija i laž i istina. Jednog dana odlučio je posetiti ostrva gde se odvijalo puno aktivnosti laganja i govorenja istine! Prvo ostrvo koje je posetio bio je Otok vitezova i podanika na kojem su oni zvani vitezovi uvek govorili istinu i oni zvani podanici uvek lažu. Dodatno znao je da je svaki stanovnik ili vitez ili podanik.

Formule koje ćemo koristiti za zapisivanje tvrđenja:

- kA - Osoba A je vitez
- $\neg\ kA$ - Osoba A je podanik
- $kA \longleftrightarrow B$ - Osoba A je rekla činjenicu B
- $kA \longleftrightarrow (Q \longleftrightarrow yes)$ - Osoba A je odgovorila sa yes na yes/no pitanje Q

Kada želimo da unapred naglasimo po kojim pretpostavkama ćemo granati dokaz (ili ako su same pretpostavke previše dugačke pa ne želimo da ih prepisujemo), možemo odmah naglasiti po kojoj formuli želimo da izvršimo grananje: *proof (cases "A")*. Ovakav zapis se suštinski ne razlikuje od dokaza u kome se koristi *proof cases* bez navođenja formule kada se naknadno instancira formula naredbom *assumes A* (odnosno *assumes* $\neg$ *A* u drugom slučaju), samo je razlika u sintaksi.

Zadatak 5.31. Svaka osoba daje odgovor *yes* na pitanje *Da li si ti vitez?*.

Kada se postavi pitanje proizvoljnom stanovniku ostrva "Da li si ti vitez?", imamo dve mogućnosti. Osoba je ili vitez ili podanik. Ako je osoba vitez ona će odgovoriti iskreno i dobićemo odgovor Da, sa druge strane ako je osoba podanik, ona će lagati i daće nam isti odgovor Da. Pitanje "Da li je k vitez?" se kodira sa $(k \longleftrightarrow yes)$, a pošto pitanje postavljamo upravo osobi k, u kombinaciji sa formulama za zapisivanje tvrđenja sa prethodne strane, možemo da formulišemo narednu lemu.

Ovaj dokaz se izvodi *rezonovanjem po slučajevima* i kada počnemo da pišemo ovaj dokaz prva naredba je *proof* (*cases k*). Tada Isabelle/HOL automatski nudi kostur ovakvog dokaza koji možemo ceo preuzeti kada kliknemo na njega (u donjem delu ekrana), ili možemo sami izabrati koji deo tog kostura nam treba pa prepisati.

lemma *no-one-admits-knave*:
assumes $k \longleftrightarrow (k \longleftrightarrow yes)$

shows *yes*
proof (*cases k*)
— Sada se dokaz grana u dve grane, jedna koja odgovara slučaju *True* (odnosno kada je pretpostavka tačna i važi *k*) i drugu koja odgovara slučaju *False* (kada je pretpostavka netačna, odnosno važi ¬ *k*). Kao i ranije, ovakve grane su razdvojene ključnom rečju next.
case *True*
from *this assms* **have** $k \longleftrightarrow yes$ **by** *auto*
from *this* ⟨*k*⟩ **show** *yes* **by** *auto*
next
case *False*
from *this assms* **have** ¬ $(k \longleftrightarrow yes)$ **by** *auto*
from *this* **have** ¬ $k \longrightarrow yes$ **by** *auto* — dodajemo među korak, koji se dokazuje sa by auto
from *this* ⟨¬ *k*⟩ **show** *yes* **by** *auto*
qed

Zadatak 5.32 (Problem 1.1.)**.** Na dan dolaska Abercrombie je sreo tri stanovnika, koje ćemo zvati *A*, *B* i *C*. Pitao je *A*: Jesi li ti vitez ili podanik? On je odgovorio, ali tako nejasno da Abercrombie nije mogao shvati što je rekao. Zatim je upitao *B*: Šta je rekao? *B* odgovori: Rekao je da je podanik. U tom trenutku, *C* se ubacio i rekao: Ne verujte u to; to je laž! Je li *C* bio vitez ili podanik?

Napomena: Prilikom formulisanja ovakvih tvrđenja, kada se od nas očekuje da sami dovršimo formulu imamo dve mogućnosti. Jedna je da logičkim razmišljanjem i eventualnom određenim transformacijama dođemo do ispravnog zaključa, a druga mogućnost je (ako smo lenji i ne želimo da razmišljamo) da iskoristimo Isabelle/HOL i da pokušamo da dokažemo različite formule. Najbrži način da ovako nešto uradimo je korišćenjem alata *sledgehammer*.

Prvo pokušavamo sa pozitivnim oblikom *kC* i vidimo da *sledgehammer* uspeva. Takođe možemo pokušati sa drugačijim zaključkom: *shows* ¬ *kC*, ali tada *sledgehammer* neće naći rešenje u zadatom vremenskom ograničenju što je dovoljno da možemo da zaključimo da je prva verzija tačna.

U ovakvim lemama prvo će u komentaru biti navedeno *sledgehammer* rešenje, a nakon toga će biti ispisan detaljan Isar dokaz.

lemma *Smullyan-1-1*:
assumes $kA \longleftrightarrow (kA \longleftrightarrow yesA)$
assumes $kB \longleftrightarrow \neg\ yesA$
assumes $kC \longleftrightarrow \neg\ kB$
shows *kC*

— sledgehammer
— Prvo pokušavamo sa *sledgehammer* i on daje sledeće rešenje: *using assms(1) assms(2) assms(3) by auto*

proof –
— Izvešćemo nekoliko međukoraka
from *assms(1)* **have** *yesA* **by** (*auto simp add*: *no-one-admits-knave*)
from *this assms(2)* **have** ¬ *kB* **by** *auto*
from *this assms(3)* **show** *kC* **by** *auto*
qed

Zadatak 5.33 (Problem 1.2.)**.** Prema drugoj verziji priče, Abercrombie nije pitao A da li je on vitez ili podanik (jer bi unapred znao koji će odgovor dobiti), već je pitao A koliko od njih trojice su bili vitezovi. Opet je A odgovorio nejasno, pa je Abercrombie upitao B što je A rekao. B je tada rekao da je A rekao da su tačno njih dvojica podanici. Tada je, kao i prije, C tvrdio da B laže. Je li je sada moguće utvrditi da li je C vitez ili podanik?

Pošto sada u postavci zadatka figuriše izjava oblika: Tačno dva od tri zadovoljavaju neko svojstvo, potrebno je uvesti narednu *definiciju* (koja je tačna samo ako su tačno dve od tri logičke vrednosti tačne). Definicije se, kao što je ranije rečeno moraju eksplicitno navesti u samom dokazu, što će u narednom dokazu biti urađeno naredbom *unfolding*.
Dodatno, naredni dokaz će biti izveden *kontradikcijom*.

definition *exactly-two* **where**
exactly-two A B C ⟷ ((*A* ∧ *B*) ∨ (*A* ∧ *C*) ∨ (*B* ∧ *C*)) ∧ ¬ (*A* ∧ *B* ∧ *C*)

Sada je lako formulisati traženo tvrđenje:

lemma *Smullyan-1-2*:
assumes *kA* ⟷ (*exactly-two* (¬ *kA*) (¬ *kB*) (¬ *kC*) ⟷ *yesA*)
assumes *kB* ⟷ *yesA*
assumes *kC* ⟷ ¬ *kB*
shows *kC*
— Isto kao i ranije, prvo testiramo sa sledgehammer i dobijamo: *using assms(1) assms(2) exactly-two-def by auto*
proof (*rule ccontr*)
assume ¬ *kC*
from *this assms* **have** *kB yesA* **by** *auto*
show *False*
proof (*cases kA*)
case *True*
from *this assms(1)* ‹*yesA*› **have** *exactly-two* (¬ *kA*) (¬ *kB*) (¬ *kC*) **by** *auto*

```
    from this ⟨kA⟩ ⟨kB⟩ ⟨¬ kC⟩ show False unfolding exactly-two-def by auto
  next
    case False
    from this assms(1) ⟨yesA⟩ have ¬ exactly-two (¬ kA) (¬ kB) (¬ kC) by auto
    from this ⟨¬ kA⟩ ⟨kB⟩ ⟨¬ kC⟩ show False unfolding exactly-two-def by auto
  qed
qed
```

Zadatak 5.34. *B* sada tvrdi da su tačno dvojica od njih vitezovi. Da li ovo menja zaključak tvrđenja?

```
lemma Smullyan-1-2':
  assumes kA ⟷ (exactly-two kA kB kC ⟷ yesA)
  assumes kB ⟷ yesA
  assumes kC ⟷ ¬ kB
  shows ¬ kC
— Isto kao i ranije, prvo testiramo sa sledgehammer i dobijamo using assms(1)
assms(2) exactly-two-def by auto
proof (rule ccontr)
— Ovaj dokaz je mogao biti napisan i samo sa proof i assume "kC" - pokušajte
  assume ¬ ¬ kC
  from this assms have kC ¬ kB ¬ yesA by auto
  show False
  proof (cases kA)
    case True
    from this ⟨¬ yesA⟩ assms(1) have ¬ exactly-two kA kB kC by auto
    from this ⟨kA⟩ ⟨¬ kB⟩ ⟨kC⟩ show False unfolding exactly-two-def by auto
  next
    case False
    from this ⟨¬ yesA⟩ assms(1) have exactly-two kA kB kC by auto
    from this ⟨¬ kA⟩ ⟨¬ kB⟩ ⟨kC⟩ show False unfolding exactly-two-def by auto
  qed
qed
```

Zadatak 5.35 (Problem 1.3.)**.** Nakon toga, Abercrombie je sreo samo dva stanovnika *A* i *B*. *A* je izjavio: "Obojica smo podanici.". Da li možemo da zaključimo šta je *A* a šta je *B*?

```
lemma Smullyan-1-3:
  assumes kA ⟷ ¬ kA ∧ ¬ kB
  shows ¬ kA ∧ kB
— using assms by auto
proof (cases kA)
```

```
  case True
  from this assms have ¬ kA ∧ ¬ kB by auto
  from this ⟨kA⟩ have False by auto
  from this show ?thesis by auto
next
  case False
  from this assms have ¬ (¬ kA ∧ ¬ kB) by auto
  from this have kA ∨ kB by auto
  from this ⟨¬ kA⟩ have kB by auto
  from this ⟨¬ kA⟩ show ?thesis by auto
qed
```

Zadatak 5.36 (Problem 1.4.)**.** Prema drugoj verziji priče, A nije rekao "Obojica smo podanici." Ono što je on rekao je "Bar jedan od nas je podanik". Ako je ova verzija odgovora tačna, šta su A i B?

```
lemma Smullyan-1-4:
  assumes kA ⟷ ¬ kA ∨ ¬ kB
  shows kA ∧ ¬ kB
— using assms by auto
proof (cases kA)
  case True
  from this assms have ¬ kA ∨ ¬ kB by auto
  from this ⟨kA⟩ have ¬ kB by auto
  from this ⟨kA⟩ show ?thesis by auto
next
  case False
  from this assms have ¬ (¬ kA ∨ ¬ kB) by auto
  from this have kA ∧ kB by auto
  from this ⟨¬ kA⟩ have False by auto
  from this show ?thesis by auto
qed
```

Zadatak 5.37 (Problem 1.5.)**.** Prema još jednoj verziji priče, A je rekao "Svi smo istog tipa: tj. ili smo svi vitezovi ili podanici." Ako je ova verzija priče tačna, šta možemo zaključiti o A i B?

```
lemma Smullyan-1-5:
  assumes kA ⟷ (kA ⟷ kB)
  shows kB
— U ovom primeru je zanimljivo da ništa ne mozemo da zaključimo o A. Pokušajte
da vidite zašto, logički, i uz pomoć Isabelle/HOL.
— using assms by blast
```

proof (*cases kA*)
  case *True*
  from *this assms* **have** $kA \longleftrightarrow kB$ **by** *auto*
  from *this* ⟨*kA*⟩ **show** *?thesis* **by** *auto*
next
  case *False*
  from *this assms* **have** $\neg\ (kA \longleftrightarrow kB)$ **by** *auto*
  from *this* ⟨$\neg$ *kA*⟩ **show** *?thesis* **by** *auto*
qed

Zadatak 5.38 (Problem 1.6.)**.** Jednom prilikom, Abercrombie je naleteo na dva stanovnika koji su se izležavali na suncu. Pitao je jednog od njih da li je onaj drugi vitez i dobio je yes/no odgovor. Nakon toga je pitao drugog stanovnika da li je prvi vitez i ponovo dobio yes/no odgovor. Da li su dobijeni odgovori obavezno isti?

Odgovore koje su dala dva stanovnika ćemo označiti sa *yesA* i *yesB* i provera da li su odgovori isti se svodi na pitanje da li važi: $yesA \longleftrightarrow yesB$.

lemma *Smullyan-1-6*:
  assumes $kA \longleftrightarrow (kB \longleftrightarrow yesA)$
  assumes $kB \longleftrightarrow (kA \longleftrightarrow yesB)$
  shows $yesA \longleftrightarrow yesB$
— using assms by auto
proof (*cases kA*)
  case *True*
  from *this assms*(*1*) **have** $kB \longleftrightarrow yesA$ **by** *auto*
  show *?thesis*
  proof (*cases kB*)
    case *True*
    from *this* ⟨$kB \longleftrightarrow yesA$⟩ **have** *yesA* **by** *auto*
    from ⟨*kB*⟩ *assms*(*2*) **have** $kA \longleftrightarrow yesB$ **by** *auto*
    from *this* ⟨*kA*⟩ **have** *yesB* **by** *auto*
    from *this* ⟨*yesA*⟩ **show** *?thesis* **by** *auto*
  next
    case *False*
    from *this* ⟨$kB \longleftrightarrow yesA$⟩ **have** $\neg$ *yesA* **by** *auto*
    from ⟨$\neg$ *kB*⟩ *assms*(*2*) **have** $\neg\ (kA \longleftrightarrow yesB)$ **by** *auto*
    from *this* ⟨*kA*⟩ **have** $\neg$ *yesB* **by** *auto*
    from *this* ⟨$\neg$ *yesA*⟩ **show** *?thesis* **by** *auto*
  qed
next
  case *False*

```
  from this assms(1) have ¬ (kB ⟷ yesA) by auto
  show ?thesis
  proof (cases kB)
    case True
    from this ‹¬ (kB ⟷ yesA)› have ¬ yesA by auto
    from ‹kB› assms(2) have kA ⟷ yesB by auto
    from this ‹¬ kA› have ¬ yesB by auto
    from this ‹¬ yesA› show ?thesis by auto
  next
    case False
    from this ‹¬ (kB ⟷ yesA)› have yesA by auto
    from ‹¬ kB› assms(2) have ¬ (kA ⟷ yesB) by auto
    from this ‹¬ kA› have yesB by auto
    from this ‹yesA› show ?thesis by auto
  qed
qed
```

Zadatak 5.39 (Problem 1.9.)**.** U sledećem susretu, Abercrombie je sreo 3 stanovnika A, B i C, koji su izjavili naredne rečenice:

A: Tačno jedan od nas je podanik.

B: Tačno dvojica od nas su podanici.

C: Svi mi smo podanici.

Ko je kog tipa?

```
definition exactly-one where
  exactly-one A B C ⟷ (A ∨ B ∨ C) ∧ ¬((A ∧ B) ∨ (A ∧ C) ∨ (B ∧ C))

lemma Smullyan-1-9:
  assumes kA ⟷ exactly-one (¬kA) (¬kB) (¬kC)
  assumes kB ⟷ exactly-two (¬kA) (¬kB) (¬kC)
  assumes kC ⟷ ¬ kA ∧ ¬ kB ∧ ¬ kC
  shows ¬ kC ∧ kB ∧ ¬ kA
— using assms unfolding exactly-one-def exactly-two-def by auto
proof (cases kC)
  case True
  from this assms(3) have ¬ kA ¬ kB ¬ kC by auto
  from this ‹kC› have False by auto
  from this show ?thesis by auto
next
  case False
  from this assms(3) have ¬ (¬ kA ∧ ¬ kB ∧ ¬ kC) by auto
  from this have kA ∨ kB ∨ kC by auto
```

```
  from this ⟨¬ kC⟩ have kA ∨ kB by auto
  from this show ?thesis
  proof
    assume kA
    from this assms(1) have exactly-one (¬kA) (¬kB) (¬kC) by auto
    from this ⟨¬ kC⟩ have kB unfolding exactly-one-def by auto
    from this assms(2) have exactly-two (¬kA) (¬kB) (¬kC) by auto
    from this ⟨¬ kC⟩ ⟨kA⟩ ⟨kB⟩ have False unfolding exactly-two-def by auto
    from this show ?thesis by auto
  next
    assume kB
    from this assms(2) have exactly-two (¬kA) (¬kB) (¬kC) by auto
    from this ⟨¬ kC⟩ ⟨kB⟩ have ¬ kA unfolding exactly-two-def by auto
    from this ⟨kB⟩ ⟨¬ kC⟩ show ?thesis by auto
  qed
qed
```

Zadatak 5.40 (Problem 1.10.)**.** Abercrombie zna da ostrvo ima poglavicu i želeo je da ga nađe. Suzio je pretragu na dvojicu braće sa imenima *Og* i *Bog*, i znao je da je jedan od njih dvojice poglavica, ali nije zao koji tačno dok nisu dali naredne izjave:

Og: *Bog* je poglavica i on je podanik.

Bog: *Og* nije poglavica, ali je vitez.

Ko je od njih dvojice poglavica?

Pošto je poglavica ili *Og* ili *Bog* i ne mogu biti poglavica istovremeno, možemo izjavu "Bog is the chief" zapisati kao "Og is not the chief".

```
lemma Smullyan-1-10:
  assumes Og ⟷ ¬ chief-Og ∧ ¬ Bog
  assumes Bog ⟷ ¬ chief-Og ∧ Og
  shows chief-Og ¬ Og ¬ Bog
  oops
```

Zbog bržeg zapisivanja preimenovaćemo promenljive u ovom dokazu i koristićemo naredni zapis umesto originalnog:

```
lemma Smullyan-1-10′:
  assumes A ⟷ ¬ C ∧ ¬ B
  assumes B ⟷ ¬ C ∧ A
  shows C ∧ ¬ A ∧ ¬ B
proof (cases A)
  case True
```

```
  from this assms(1) have ¬ C ∧ ¬ B by auto
  from this have ¬ C ¬ B by auto
  from this assms(2) have ¬ (¬ C ∧ A) by auto
  from this have C ∨ ¬ A by auto
  from this ⟨A⟩ have C by auto
  from this ⟨¬ C⟩ have False by auto
  from this show ?thesis by auto
next
  case False
  from this assms(1) have ¬ (¬ C ∧ ¬ B) by auto
  from this have C ∨ B by auto
  from this show ?thesis
  proof
    assume C
    from this ⟨¬ A⟩ assms(2) have ¬ B by auto
    from this ⟨¬ A⟩ ⟨C⟩ show ?thesis by auto
  next
    assume B
    from this assms(2) have ¬ C ∧ A by auto
    from this have ¬ C A by auto
    from this ⟨¬ A⟩ have False by auto
    from this show ?thesis by auto
  qed
qed
```

Zadatak 5.41 (Problem 1.11.)**.** Zamislimo da ste posetili Ostrvo vitezova i podanika jer ste čuli za glasinu da na ostrvu postoji zakopano blago. Sretnete stanovnika tog ostrva i želite da saznate da li stvarno postoji zlato na ostrvu, ali ne znate da li je on vitez ili podanik. Dozvoljeno je postaviti samo jedno yes/no pitanje. Kako treba da glasi to pitanje?

Pitanje koje treba postaviti je $k \longleftrightarrow g$ tj. da li si ti vitez akko postoji zlato.

```
lemma NelsonGoodman:
  shows (k ⟷ (k ⟷ g)) ⟷ g
  by auto

lemma Smullyan-1-11:
  assumes k ⟷ ((k ⟷ g) ⟷ yes)
  shows yes ⟷ g
— using assms by auto
proof (cases k)
```

```
  case True
  from this assms have (k ⟷ g) ⟷ yes by auto
  show ?thesis
  proof (cases g)
    case True
    from this ⟨k⟩ ⟨(k ⟷ g) ⟷ yes⟩ have yes by auto
    from this ⟨g⟩ show ?thesis by auto
  next
    case False
    from this ⟨k⟩ ⟨(k ⟷ g) ⟷ yes⟩ have ¬ yes by auto
    from this ⟨¬ g⟩ show ?thesis by auto
  qed
next
  case False
  from this assms have ¬ ((k ⟷ g) ⟷ yes) by auto
  show ?thesis
  proof (cases g)
    case True
    from this ⟨¬ k⟩ ⟨¬ ((k ⟷ g) ⟷ yes)⟩ have yes by auto
    from this ⟨g⟩ show ?thesis by auto
  next
    case False
    from this ⟨¬ k⟩ ⟨¬ ((k ⟷ g) ⟷ yes)⟩ have ¬ yes by auto
    from this ⟨¬ g⟩ show ?thesis by auto
  qed
qed
```

Zadatak 5.42 (Problem 1.12.)**.** Na narednom ostrvu svi vitezovi žive u jednom selu, a svi podanici u drugom. Posetilac ostrva stoji na raskršću dva puta od kojih jedan vodi u selo vitezova a drugi u selo podanika. Želi da ode u selo vitezova, ali ne zna kojim putem treba da krene. Jedan od stanovnika ostrva stoji na raskršću i posetilac može da mu postavi jedno pitanje na koje može dobiti yes/no odgovor. Jedan način jeste da se upotrebi Nelson Goodman princip i postavi pitanje: "Da li si ti vitez akko levi put vodi u selo vitezova?" Međutim, postoji mnogo jednostavnije i prirodnije pitanje koje sadrži samo 8 reči. Da li možete da otkrijete takvo pitanje?

Pitanje: da li levi put vodi u tvoje selo?

Analiziraćemo ovo pitanje i njegove odgovore detaljno. Zamislimo da odgovori sa *yes*. Ako je on vitez, onda levi put stvarno vodi do njegovog sela, tj. do sela vitezova. Ako je on podanik, onda levi put ne vodi do njegovog sela, koje je selo podanik, pa onda vodi do sela vitezova. Znači u svakom slučaju odgovor *yes* znači da posetilac treba da krene levim putem.

Pretpostavimo da je odgovorio *no*. Ako je on vitez, onda levi put stvarno ne vodi do njegovog sela, pa onda desni put vodi do njegovog sela tj. do sela vitezova. Ako je on podanik, onda levi put vodi do njegovog sela (s obzirom da je on rekao da ne vodi), što je selo podanika, pa otuda desni put vodi do sela vitezova. Znači, u svakom slučaju odgovor *no* znači da posetilac treba da krene desnim putem.

lemma *Smullyan-1-12*:
assumes $k \longleftrightarrow (((k \wedge l) \vee (\neg\ k \wedge \neg\ l)) \longleftrightarrow yes)$
shows $yes \longleftrightarrow l$
— using assms by auto
proof (*cases k*)
case *True*
from *this assms* **have** $*$: $((k \wedge l) \vee (\neg\ k \wedge \neg\ l)) \longleftrightarrow yes$ **by** *auto*
from ⟨*k*⟩ **have** $**$: $k \wedge l \longleftrightarrow l$ **by** *auto*
from ⟨*k*⟩ **have** $\neg\ (\neg\ k \wedge \neg\ l)$ **by** *auto*
from *this* $*$ **have** $k \wedge l \longleftrightarrow yes$ **by** *auto*
from *this* $**$ **show** *?thesis* **by** *auto*
next
case *False*
from *this assms* **have** $*$: $\neg\ (((k \wedge l) \vee (\neg\ k \wedge \neg\ l)) \longleftrightarrow yes)$ **by** *auto*
from ⟨$\neg\ k$⟩ **have** $**$: $\neg\ k \wedge \neg\ l \longleftrightarrow \neg\ l$ **by** *auto*
from ⟨$\neg\ k$⟩ **have** $\neg\ (k \wedge l)$ **by** *auto*
from *this* $*$ **have** $\neg\ ((\neg\ k \wedge \neg\ l) \longleftrightarrow yes)$ **by** *auto*
from *this* $**$ **have** $\neg\ ((\neg\ l) \longleftrightarrow yes)$ **by** *auto*
from *this* **have** $(\neg\ l \wedge \neg\ yes) \vee (l \wedge yes)$ **by** *auto*
from *this* **show** *?thesis* **by** *auto*
qed

Na sledećem ostrvu Abercrombie je primetio da i žene mogu da budu vitezovi i podanici. Ono što je zanimljivo je da za razliku od muškaraca kod kojih vitezovi govore istinu, a podanici lažu, žene su radile obratno: Ženski podanici su govorili istinu a ženski vitezovi lažu.

Zadatak 5.43 (Problem 2.1.)**.** Abercrombie je razgovarao sa stanovnikom koji je stajao daleko na bini i nosio je masku. Želeo je da sazna da li je stanovnik žena ili muškarac. Bilo mu je dozvoljeno da postavi samo jedno *yes/no* pitanje na koje bi onda stanovnik mogao da da odgovor na tabli (kako ne bi odao boju svoga glasa).

Odgovarajuće pitanje: Da li si ti vitez?

Svaka muška osoba bi odgovorila *yes*, na osnovu leme *no-one-admits-knave*: *assumes* $k \longleftrightarrow (k \longleftrightarrow yes)$ *shows yes*, koju smo već dokazali.

Svaka ženska osoba bi odgovorila *no*. Svaka ženska osoba može biti ili vitez ili podanik. Ako je ženska osoba vitez ona će lagati i dobićemo odgovor *no*, sa druge strane ako je osoba podanik, ona će govoriti istinu i daće nam odgovor *no*.

Otuda odgovor *yes* na ovo pitanje znači da je u pitanju *male* (zadovoljeno m); odgovor *no*, znaci da je u pitanju *female* (nije zadovoljeno m); Otuda možemo da zaključimo da važi ekvivalencija $yes \longleftrightarrow m$.

Sada kada k može biti muško i žensko, dodajemo u lemu *no-one-admits-knave* i tu pretpostavku, da je k upravo muško: *assumes* $(k \wedge m) \longleftrightarrow (k \longleftrightarrow yes)$ *shows yes*.

Napomena: samostalno, ovo nije valjana lema, već samo opis kako od već postojećih tvrđenja možemo doći do formule koja nam treba za opis početnog problema.

Svaka ženska osoba, analogno, na ovo pitanje bi odgovorila sa *no*, tj. $\neg$ *yes*. Prvo možemo zameniti k sa leve strane sa $\neg\ k$ (sa desne strane ostaje k zato što je to odgovor na pitanje "da li si ti vitez"). A nakon toga možemo dodati pretpostavku da je u pitanju ženska osoba (odnosno $\neg\ m$): *assumes* $(\neg\ k \wedge \neg\ m) \longleftrightarrow (k \longleftrightarrow yes)$ *shows* $\neg$ *yes*

Sada možemo spojiti ova dva tvrđenja u jedno:

lemma *Smullyan-2-1*:
  assumes $(m \wedge k) \vee (\neg\ m \wedge \neg\ k) \longleftrightarrow (k \longleftrightarrow yes)$
  shows $yes \longleftrightarrow m$
proof (*cases* m)
  case *True*
  show $yes \longleftrightarrow m$
  proof (*cases* k) — muško i vitez - govori istinu
    case *True*
    from ‹k› ‹m› *assms* **have** $k \longleftrightarrow yes$ **by** *auto*
    from *this* ‹k› ‹m› **show** $yes \longleftrightarrow m$ **by** *auto*
  next
    case *False* — muško i nije vitez - laže
    from ‹$\neg\ k$› ‹m› *assms* **have** $\neg\ (k \longleftrightarrow yes)$ **by** *auto*
    from *this* ‹$\neg\ k$› **have** *yes* **by** *auto*
    from *this* ‹m› **show** $yes \longleftrightarrow m$ **by** *auto*
  qed
next
  case *False* — žensko
  show $yes \longleftrightarrow m$
  proof (*cases* k) — žensko i vitez - laže
    case *True*

```
  from ⟨k⟩ ⟨¬ m⟩ assms have ¬ (k ⟷ yes) by auto
  from this ⟨k⟩ have ¬ yes by auto
  from this ⟨¬ m⟩ show yes ⟷ m by auto
 next
  case False — žensko i nije vitez - govori istinu
  from ⟨¬ k⟩ ⟨¬ m⟩ assms have k ⟷ yes by auto
  from this ⟨¬ k⟩ have ¬ yes by auto
  from this ⟨¬ m⟩ show yes ⟷ m by auto
 qed
qed
```

Zadatak 5.44 (Problem 2.2.)**.** Stanovnik ostrva je napustio binu i novi maskirani stanovnik se pojavio. Ovaj put Abercrombie nije želeo da sazna pol stanovnika već da li je stanovnik vitez ili podanik. Koje pitanje treba da postavi?

Pitanje: Da li si ti muško?

```
lemma Smullyan-2-2:
 assumes (m ⟷ k) ⟷ (m ⟷ yes)
 shows yes ⟷ k
 using assms
 by blast
```

Zadatak 5.45. (Problem 2.3.) Next, a new inhabitant appeared on the stage, and Abercrombie was to determine whether or not the inhabitant was married, but he could ask only one yes/no question. What question should he ask?

```
lemma Smullyan-2-3:
 assumes (m ⟷ k) ⟷ (((m ⟷ k) ⟷ married) ⟷ yes)
 shows yes ⟷ married
 using assms
 by blast
```

Zadatak 5.46 (Problem 2.4.)**.** U sledećem testu, stanovnik ostrva je napisao rečenicu iz koje Abercrombie može direktno da zaključi da je u pitanju ženski vitez. Koja rečenica bi ovo mogla da uradi?

Pitanje: Da li si ti muškarac i podanik?

```
lemma Smullyan-2-4:
 assumes (m ⟷ k) ⟷ ((m ∧ ¬k) ⟷ yes)
```

shows *yes* $\longleftrightarrow \neg\ m \wedge k$
using *assms*
by *blast*

Zadatak 5.47 (Problem 2.5.)**.** Naredna rečenica koja je bila zapisana je pomogla Abercromibiju da zaključi da je stanovnik koji ju je napisao muški vitez. Koja rečenica je u pitanju?

Pitanje: da li si ti muško ili vitez?

lemma *Smullyan-2-5*:
assumes $(m \longleftrightarrow k) \longleftrightarrow ((m \vee k) \longleftrightarrow yes)$
shows *yes* $\longleftrightarrow m \wedge k$
using *assms*
by *blast*

Zadatak 5.48 (Problem 8.1.)**.** Zamislite da upoznate stanovnika ostrva i postavite mu pitanje da li postoji zlato na ostrvu. On odgovori ovako: "Ako sam ja vitez, onda postoji zlato na ovom ostrvu". Da li je moguće iz njegovog odgovora odrediti da li postoji zlati i da li je on vitez ili podanik?

U ovakvim zadacima, ako niste sigurni kako zapisati tvrđenje možete koristiti *sledgehammer* pa istestirati tvrđenja.

lemma *Smullyan-8-1*:
assumes $k \longleftrightarrow (k \longrightarrow g)$
shows $g \wedge k$
proof (*cases k*)
case *True*
from *this assms* **have** $k \longrightarrow g$ **by** *auto*
from *this* ‹*k*› **show** $g \wedge k$ **by** *auto*
next
case *False*
from *this assms* **have** $\neg\ (k \longrightarrow g)$ **by** *auto*
from *this* **have** $\neg\ (\neg\ k \vee g)$ **by** *auto*
from *this* **have** $k \wedge \neg\ g$ **by** *auto*
from *this* ‹$\neg$ *k*› **have** *False* **by** *auto*
from *this* **show** $g \wedge k$ **by** *auto*
qed

Zadatak 5.49 (Problem 8.3.)**.** Pretpostavimo da je stanovnik rekao: "Ja sam podanik i ne postoji zlato na ostrvu". Šta može biti zaključeno iz ovoga?

U zapisu ovog tvrđenja, slično kao i ranije, možemo ili sami shvatiti šta je traženi zaključak, ili nasumice ređati razne zaključke pa videti koje možemo da elimišemo. U ovom slučaju, ako pokušamo da izvedemo samo k Isabelle/HOL će naći kontraprimer, što znači da mora važiti $\neg\ k$. Slično možemo zaključiti ako pokušamo da izvedemo $\neg\ g$. Otuda vidimo da je ispravna sledeća formulacija:

lemma *Smullyan-8-3*:
assumes $k \longleftrightarrow (\neg\ k \wedge \neg\ g)$
shows $\neg\ k \wedge g$
— using assms by auto - ovo bi bio automatski dokaz, a sada ga treba raspisati u Isar-u
proof (*cases k*)
case *True*
from *this assms* **have** $\neg\ k \wedge \neg\ g$ **by** *auto*
from *this* **have** $\neg\ k$ **by** *auto*
from *this* ‹*k*› **have** *False* **by** *auto* — nakon što izvedemo kontradikciju, možemo izvesti bilo koji zaključak
from *this* **show** *?thesis* **by** *auto*
next
case *False*
from *this assms* **have** $\neg\ (\neg\ k \wedge \neg\ g)$ **by** *auto*
from *this* **have** $k \vee g$ **by** *auto*
from *this* ‹$\neg\ k$› **have** g **by** *auto*
from *this* ‹$\neg\ k$› **show** *?thesis* **by** *auto*
qed

Kvantifikatori

Zadatak 5.50. Ako svi tvrde isto, onda su svi istog tipa (ili su svi vitezovi ili su svi podanici).

lemma *all-say-the-same*:
assumes $\forall\ x.\ k\ x \longleftrightarrow P$
shows $(\forall\ x.\ k\ x) \vee (\forall\ x.\ \neg\ k\ x)$
proof (*rule ccontr*)
assume $\neg$ *?thesis*
from *this* **have** $(\exists\ x.\ k\ x) \wedge (\exists\ x.\ \neg\ k\ x)$ **by** *auto*
from *this* **obtain** *kv kn* **where** $k\ kn\ \neg\ k\ kv$ **by** *auto*
from *this assms* **have** $P\ \neg\ P$ **by** *auto*
from *this* **show** *False* **by** *auto*
qed

Zadatak 5.51 (Problem 12.1.)**.** Na ostrvu koje je posetio svi stanovnici su izjavili istu stvar: "Svi mi smo istog tipa". Šta može biti zaključeno o stanovnicima ovog ostrva?

lemma *Smullyan-12-1*:
assumes $\forall\ x.\ (k\ x \longleftrightarrow (\forall\ x.\ k\ x) \vee (\forall\ x.\ \neg\ k\ x))$
shows $\forall\ x.\ k\ x$
— sledgehammer *using assms by blast*
proof (*rule ccontr*)
assume $\neg$ *?thesis*
from *this* **have** $\exists\ x.\ \neg\ k\ x$ **by** *auto*
from *this* **obtain** *kx1* **where** $\neg\ k$ *kx1* **by** *auto*
from *this assms* **have** $*$: $\neg\ ((\forall\ x.\ k\ x) \vee (\forall\ x.\ \neg\ k\ x))$ **by** *blast*
from *this* **have** $\exists\ x.\ k\ x$ **by** *auto*
from *this* **obtain** *kx2* **where** k *kx2* **by** *auto*
from *this assms* **have** $**$: $(\forall\ x.\ k\ x) \vee (\forall\ x.\ \neg\ k\ x)$ **by** *blast*
from $*$ $**$ **show** *False* **by** *auto*
qed

Zadatak 5.52 (Problem 12.2.)**.** Na sledećem ostrvu svi stanovnici su rekli: "Neki od nas su vitezovi a neki od nas su podanici". Šta može biti zaključeno iz ovoga?

lemma *Smullyan-12-2*:
assumes $\forall\ x.\ (k\ x \longleftrightarrow (\exists\ x.\ k\ x) \wedge (\exists\ x.\ \neg\ k\ x))$
shows $\forall\ x.\ \neg\ k\ x$
— sledgehammer using assms by blast
proof (*rule ccontr*)
assume $\neg$ *?thesis*
from *this* **have** $\exists\ x.\ \neg\ \neg\ k\ x$ **by** *auto*
from *this* **obtain** *kx* **where** k *kx* **by** *auto*
from *this assms* **have** $*$: $(\exists\ x.\ k\ x) \wedge (\exists\ x.\ \neg\ k\ x)$ **by** *blast* — by auto ne prolazi
from *this* **have** $\exists\ x.\ k\ x\ \exists\ x.\ \neg\ k\ x$ **by** *auto*
from *this* **obtain** *kx1 kx2* **where** k *kx1* $\neg\ k$ *kx2* **by** *auto*
from *this assms* **have** $**$: $\neg\ ((\exists\ x.\ k\ x) \wedge (\exists\ x.\ \neg\ k\ x))$ **by** *blast*
from $*$ $**$ **show** *False* **by** *auto*
qed

Zadatak 5.53 (Problem 12.3.)**.** Na sledećem ostrvu, Abercrombie je razgovarao sa svim stanovnicima ostrva osim sa jednim koji je spavao. Svi su rekli isto: "Svi mi smo podanici". Sledećeg dana sreo je uspavanog stanovnika i pitao ga: "Da li je istina da su svi stanovnici ovog ostrva podanici?" On je dao *yes/no* odgovor na ovo pitanje. Koji je odgovor dao?

lemma *Smullyan-12-3*:
assumes $\forall\ x.\ x \neq sleepy \longrightarrow (k\ x \longleftrightarrow (\forall\ x.\ \neg\ k\ x))$
assumes $k\ sleepy \longleftrightarrow ((\forall\ x.\ \neg\ k\ x) \longleftrightarrow yes)$
shows $\neg\ yes$
— sledgehammer *using assms(1) assms(2) by auto*
proof (*rule ccontr*)
assume $\neg\ \neg\ yes$
from *this* **have** *yes* **by** *auto*
from *this* **show** *False*
proof (*cases k sleepy*)
case *True*
from *this assms(2)* ‹*yes*› **have** $\forall\ x.\ \neg\ k\ x$ **by** *auto*
from *this* **have** $\neg\ k\ sleepy$ **by** *auto*
from *this True* **show** *False* **by** *auto*
next
case *False*
from *this assms(2)* ‹*yes*› **have** $\neg\ (\forall\ x.\ \neg\ k\ x)$ **by** *auto*
from *this* **have** $\exists\ x.\ k\ x$ **by** *auto*
from *this* **obtain** *kx* **where** *k kx* **by** *auto*
from *this False* **have** $kx \neq sleepy$ **by** *auto*
from *this assms(1)* ‹*k kx*› **have** $\forall\ x.\ \neg\ k\ x$ **by** *auto*
from *this* ‹*k kx*› **show** *False* **by** *auto*
qed
qed

end

6

Brojevi i matematička indukcija

theory *Cas8-vezbe*
imports *Complex-Main*
begin

6.1 Tip prirodnih brojeva

U Isabelle/HOL postoji ugrađen tip prirodnih brojeva koji se označava sa *nat*. Tip prirodnih brojeva je rekurzivno definisan tip sa konstruktorima nula i sledbenik. Posebno se dobro slaže sa induktivnim dokazima i primitivno rekurzivnim definicijama.

Teorija prirodnih brojeva moze se naći na: https://isabelle.in.tum.de/library/HOL/HOL/Nat.html

Ključnu reč *term* možemo koristiti da proverimo koji je tip određenog izraza. Na primer, ako zapišemo samo *3* - to je vrednost proizvoljnog, nepoznatog tipa; ako zapišemo *3*::*nat* - to je vrednost koja pripada skupu prirodnih brojeva. Treba napomenuti da i nula (*0*::*nat*) spada u skup prirodnih brojeva u Isabelle/HOL. Sa *int* označavamo tip celih brojeva. I celi i prirodni brojevi su u Isabelle/HOL kodirani tako da nemaju ograničenje.

typ *nat*

term *3*
term *3*::*nat*
term *0*::*nat*
term -3::*int*

Operacije nad prirodnim brojevima se zapisuju na uobičajeni način: $+$, $-$, $*$, *div*, *mod*. Pored ovih operacija bitna je i operacija *Suc* koja nam daje sledbenik prirodnog broja i biće korišćena kod indukcije.

Primer dokaza u Isabelle/HOL nad prirodnim brojevima se može videti u nastavku teksta (ovaj dokaz nećemo objašnjavati sada). Prilikom definisanja leme, komandom *fixes* se može naglasiti da se lema odnosi samo na prirodne brojeve. Bez te pretpostavke, naredni dokaz ne bi bio uspešan.

lemma
  fixes $a\ b\ c\ d$ :: *nat*
  shows $a * ((b + c) + d) = (a * b + a * d) + a * c$
proof–
  have $a * ((b + c) + d) = a * (b + c) + a * d$
    by (*rule distrib-left*)
  also have ... $= (a * b + a * c) + a * d$
    by (*simp add*: *add-mult-distrib2*)

also have ... $= a * b + (a * c + a * d)$
by *simp*
also have ... $= a * b + (a * d + a * c)$
by *simp*
also have ... $= (a * b + a * d) + a * c$
by *simp*
finally
show *?thesis*
by *simp*
qed

U Isabelle/HOL već ostoji dosta teorema nad prirodnim brojevima koje možemo koristiti po potrebi. Za pretraživanje skupa teorema možemo koristiti naredbu `find_theorems` koja nam nudi pretragu po imenu (uz ključnu reč `name`, ili pretragu po šablonu koji nam treba. Na primer, uz pomoć narednih dveju naredbi možemo pronaći teoreme *Suc-eq-plus1*: *Suc n = n + 1* i *add-mult-distrib2*: $k * (m + n) = k * m + k * n$.

find-theorems *name*: *Suc-eq*
thm *Suc-eq-plus1* — *Suc n = n + 1*

find-theorems - * (- + -) = - * - + - * -
thm *add-mult-distrib2* — $k * (m + n) = k * m + k * n$

U ovom poglavlju, prilikom pisanja Isar dokaza, od automatskih alata koristićemo samo *simp* kada radimo sa brojevima. Ako želimo da pored simplifikatora koristimo još i pravilo A, to ćemo naglasiti naredbom *by* (*simp add*: *rule A*). Ako želimo da simplifikator koristi iskučivo pravilo A (i ni jedno drugo pravilo), to ćemo naglasiti naredbom *by* (*simp only*: *rule A*).

6.2 Matematička indukcija

U ovom poglavlju ćemo detaljno prikazati primenu metoda matematičke indukcije u okviru Isabelle/HOL i Isar dokaza.

Zadatak 6.1. Dokazati da važi: $0 + 1 + 2 + ... + (n - 1) = n * (n - 1)/2$.

Za početak je potrebno definisati funkciju koja računa sumu brojeva od 0 do n. Vrednosti te funkcije redom za vrednosti 0, 1, 2, 3, i 4 su brojevi 0, 1, 3, 6, 10,.... Ovi brojevi se nazivaju trougaoni brojevi.

Koristićemo **rekurzivnu definiciju** za zapis ove funkcije. Rekurzivne definicije nad tipom prirodnih brojeva su direktno povezane sa načinom

na koji su prirodni brojevi definisani. Svaki prirodni broj je ili 0 ili je sledbenik nekog drugog prirodnog broja. Kada se definišu rekurzivne funkcije moramo voditi računa o dve stvari: izlaz iz rekurzije (za prirodne brojeve to će biti broj 0) i rekurzivni poziv koji se ovde definiše za parametar koji je nečiji sledbenik. Za definiciju se koristi ključna rec `primrec` i odnosi se na primitivnu rekurziju i razlikuje dva slučaja, slučaj nule i slučaj sledbenika. Nakon imena funkcije zadaje se njen tip, ključna reč *where* i nakon toga definicija funkcije.

primrec *trougaoni* :: *nat* ⇒ *nat* **where**
trougaoni 0 = 0 |
trougaoni (*Suc n*) = *trougaoni n* + *Suc n*
— sa leve strane oko Suc n je neophodna zagrada, a sa desne strane nije

value *trougaoni 5* — vrednost 15

Mnogi dokazi nad prirodnim brojevima se mogu dokazati samo koristeći metod matematičke indukcije. Pored automatskog dokazivača *auto* koristi se i metod matematičke indukcije *induction* koji se mora eksplicitno pozvati i za koji moramo navesti po kojoj promenljivoj izvodimo taj dokaz.

Naglasimo još da je teorema u originalnom obliku zapisana za vrednost *n−1* sa leve strane a u narednoj formulaciji imamo vrednost *n* sa leve strane pa u skladu sa tim je promenjena i vrednost sa desne strane.

lemma *trougaoni n* = *n* ∗ (*n* + *1*) *div 2*
by (*induction n*) *auto*

Međutim, pogledaćemo kako izgleda detaljan dokaz ovog tvrđenja zapisan u jeziku Isar.

U okviru dokaza koji koriste matematičku indukciju moramo razlikovati funkciju, odnosno *term Suc* koji se koristi za zapisivanje sledbenika celog broja, od induktivne pretpostavke koja se zapisuje istim imenom Suc i čiji izgled se može dobiti naredbom *thm Suc*. Otuda sam term *Suc* može da se pojavi kao deo formula koje se javljaju u dokazu, dok teorema *Suc* može da se pojavi kao deo metoda koji se koristi za dokazivanje tekućeg koraka. Otuda se naredba *using Suc* odnosi na korišćenje induktivne hipoteze i u svakoj novoj teoremi će imati oblik odgovarajuće induktivne pretpostavke (koju dobijamo na osnovu tvrđenja teoreme).

Napomena: ako pogledate u Isabelle/HOL ispis naredbe `thm Suc` dobićete ispis teoreme sa imenom *Suc*. Ovakav ispis se može dobiti samo u okviru dokaza koji u sebi koristi indukciju po skupu prirodnih brojeva i van nekog takvog dokaza teorema sa tim imenom ne postoji.

lemma
*trougaoni n = n * (n + 1) div 2*
proof (*induction n*)
case *0*
then show *?case*
by *simp*
next
case (*Suc n*)
term *Suc*
thm *Suc* — Ispis značenja ove teoreme u ovom konkretnom dokazu *trougaoni n = n * (n + 1) div 2*
note *ih = this*
have *trougaoni (Suc n) = trougaoni n + Suc n*
by *simp*
also have *... = (n * (n + 1)) div 2 + Suc n*
using *ih*
by *simp*
also have *(n * (n + 1)) div 2 + Suc n = (n + 1) * (n + 2) div 2*
by *simp*
finally
show *?case*
by *simp*
qed

Naredni primeri predstavljaju rešenja zadataka preuzetih iz materijala profesora Nebojše Ikodinovića: http://www.matf.bg.ac.rs/p/files/43-matematicka_indukcija.pdf.

U dokazima će nam često trebati teoreme vezane za sabiranje, oduzimanje i množenje. Nekoliko najčešće korišćenih se nalaze u skupu teorema sa imenom `algebra_simps`. Teoreme vezane za deljenje se nalaze u posebnom skupu teorema koji se zove `field_simps` (ovaj skup je nadskup prethodnog skupa).

thm *algebra-simps*

thm *field-simps*

Kada se neka od potrebnih lema, ili ceo skup lema, dodaje automatskim alatima ključne reči koje se koriste za metod *auto* su *simp add* i *simp del*.

Zadatak 6.2. Dokazati da važi $1 + \ldots + n = n * (n + 1)/2$. [1]

[1] Zadatak 1 iz dodatnog materijala.

Prvo je potrebno uz pomoć primitivne rekurzije definisati funkciju koja računa izraz sa leve strane jednakosti koju treba da dokažemo:

```
primrec zbir-do :: nat ⇒ nat where
zbir-do 0 = 0 |
zbir-do (Suc n) = Suc n + (zbir-do n)
```

Sada uz pomoć automatskih alata, i skupa lema `algebra_simps` možemo dokazati narednu lemu.

Deljenje nad celim brojevima se u Isabelle/HOL zapisuje sa *div*, pa se traženo tvrđenje može zapisati i automatski dokazati na sledeći način:

```
lemma suma-do-n-automatski: zbir-do n = n * (n+1) div 2
by (induction n, auto simp add: algebra-simps)
```

Dokaz iste leme u Isar-u:

```
lemma suma-do-n-isar: zbir-do n = n * (n+1) div 2
proof (induction n)
case 0
  show ?case
    by (simp only: zbir-do.simps(1))
next
case (Suc n)
thm Suc
  show ?case
  proof −
    have zbir-do (Suc n) = Suc n + zbir-do (n)
      by (simp only: zbir-do.simps(2))
    also have ... = Suc n + n * (n+1) div 2
    using Suc by simp
    also have ... = 2 * (Suc n) div 2 + (Suc n) * n div 2
    by simp
    also have ... = (Suc n) * (2 + n) div 2
    by simp
    finally show ?thesis by simp
  qed
qed
```

Zadatak 6.3. Dokazati da važi $1 + 3 + 5 + \ldots + (2 * n - 1) = n * n$. [2]

```
value 1 + 3 + 5 + 7 + 9::nat
value 5*5::nat
```

[2] Zadatak 2 iz dodatnog materijala.

U ovom zadatku prvo je potrebno definisati funkciju sa leve strane date jednačine. U pitanju je zbir prvih n neparnih brojeva, što možemo definisati primitivnom rekurzijom na sledeći način:

```
primrec zbir-neparnih :: nat ⇒ nat where
zbir-neparnih 0 = 0 |
zbir-neparnih (Suc n) = 2∗(Suc n) − 1 + (zbir-neparnih n)
```

Sada možemo dokazati traženu lemu i automatski i raspisano u Isar-u.

```
lemma neparni-auto: zbir-neparnih n = n∗n
by (induction n, auto simp add: algebra-simps)
```

Dokaz iste leme u Isar-u:

```
lemma neparni-isar: zbir-neparnih n = n∗n
proof (induction n)
  case 0
  show ?case by (simp only: zbir-neparnih.simps(1))
next
  case (Suc n)
  show ?case
  proof −
    have zbir-neparnih (Suc n) = 2∗(Suc n) − 1 + zbir-neparnih n
      by (simp only: zbir-neparnih.simps(2))
    also have ... = 2∗(Suc n) − 1 + n∗n using Suc by simp
    also have ... = 2∗n + 1 + n∗n by simp
    finally show ?thesis by simp
  qed
qed
```

Zadatak 6.4. Dokazati da važi: $-1+3-5+\ldots+(-1)^n*(2*n-1) = (-1)^n*n$. [3]

Prvi korak je definisanje funkcije koja opisuje izraz sa leve strane:

```
primrec zbir-naizmenicno:: nat ⇒ int where
zbir-naizmenicno 0 = 0 |
zbir-naizmenicno (Suc n) = zbir-naizmenicno n + (−1) ^(Suc n) ∗ (2 ∗ (Suc n)
− 1)
```

Kada se radi sa celim brojevima, oduzimanje može u nekim situacijama napraviti problem i biće potrebna konverzija u ceo broj, na primer: *int*(*Suc n*).

[3] Zadatak 5 iz dodatnog materijala.

Nakon raspisivanja dokaza na papiru, vidimo da su nam potrebne neke teoreme određenog formata. Pretragu po formatu smo već koristili:

find-theorems - ^ *Suc* -
find-theorems - * (- + -) = - * - + - * -
thm *int-distrib*

lemma *zad5*: *zbir-naizmenicno n* = (*−1*) ^*n* * *n*
proof (*induction n*)
case *0*
then show *?case*
by *simp*
next
case (*Suc n*)
show *?case*
proof −
have *zbir-naizmenicno* (*Suc n*) = *zbir-naizmenicno* (*n*) + (*−1*) ^(*Suc n*) * (*2* * (*Suc n*) − *1*)
by (*simp only*: *zbir-naizmenicno.simps*)
also have ... = (*−1*) ^*n* * *n* + (*−1*) ^(*n+1*)*(*2*(*n+1*)*−1*)
using *Suc* **by** *simp*
also have ... = (*−1*) ^*n* *(*−1*)*(*−1*)*n* + (*−1*) ^(*n+1*)*(*2*n+1*)
by *simp*
also have ... = (*−1*) ^(*n+1*)*(*−1*)*n* + (*−1*) ^(*n+1*)*(*2*n+1*)
by *simp*
also have ... = (*−1*) ^(*n+1*) * ((*−1*)*n* + (*2*n+1*))
by (*simp add*: *int-distrib*) — ili *using int-distrib by simp*
also have ... = (*−1*) ^(*n+1*) * (*n+1*) **by** *simp*
finally show *?thesis* **by** *simp*
qed
qed

6.2.1 Rad sa matricama

U narednom primeru ćemo pokazati množenje matrica u Isabelle/HOL. Prilikom zapisivanja matrice iskoristićemo to što je, u ovom primeru, dimenzija matrice fiksna pa je najjednostavnije da matricu predstavimo uređenom četvorkom njenih elemenata. Elemente pišemo po vrstama i eksplicitno navodimo tip ove matrice i koristimo operator × (koji se zapisuje sa `\<times>`) pomoću kog kreiramo tip uređenog para, odnosno uređene n-torke.

term (*1*, *1*, *0*, *1*) :: *nat* × *nat* × *nat* × *nat*

Da ne bismo stalno navodili ovu četvorku, možemo koristiti sinonime (skraćenice) za već postojeće tipove. Oni se raspisuju nakon parsiranja teksta i nisu deo interne reprezentacije niti ispisa. Sinonimi služe udobnosti zapisa korisnika. Sinonim kojim uvodimo matricu dimenzije 2×2 ćemo uvesti na sledeći način:

type-synonym *mat2* = *nat* × *nat* × *nat* × *nat*
term (*1*, *1*, *0*, *1*) :: *mat2*

Zadatak 6.5. Dokazati da važi $(1, 1, 0, 1)^n = (1, n, 0, 1)$. [4]

Sledeći korak je da definišemo sve pojmove koji su nam se javljali u dokazu iz udžbenika: jediničnu matricu, množenje matrica, stepenovanje matrica.

Jediničnu matricu uvodimo narednom definicijom:

definition *eye* :: *mat2* **where**
eye = (*1*, *0*, *0*, *1*)

Nakon ovoga definišemo proizvod dve matrice i uvodimo novi način definisanja objekata. Definicija (odnosno ključna reč *definition*) ne može da se koristi na ovom mestu zato što argument funkcije koja se uvodi definicijom mora biti jedna promenljiva (na primer *mat-mul m1 m2*, nakon čega bi bilo potrebno da mi sami nekim novim operatorima izdvajamo element po element). Zbog toga ovde koristimo ključnu reč *fun* koja koristi uklapanje šablona *(pattern matching)*. Sada umesto da sa leve strane imamo samo imena promenljivih, imamo mogućnost da na tom mestu zadamo stukturu vrednosti koje te promenljive predstavljaju.

fun *mat-mul* :: *mat2* ⇒ *mat2* ⇒ *mat2* **where**
mat-mul (*a1*, *b1*, *c1*, *d1*) (*a2*, *b2*, *c2*, *d2*) = (*a1*∗*a2* + *b1*∗*c2*, *a1*∗*b2* + *b1*∗*d2*, *c1*∗*a2* + *d1*∗*c2*, *c1*∗*b2* + *d1*∗*d2*)

U buduće ćemo *fun* koristiti za definisanje funkcija običnom rekurzijom.

Sada definišemo stepenovanje matrica korišćenjem primitivne rekurzije. I korišćenjem funkcije *value* proveravamo da li je definicija dobra.

primrec *mat-pow* :: *mat2* ⇒ *nat* ⇒ *mat2* **where**
mat-pow A 0 = *eye* |
mat-pow A (*Suc n*) = *mat-mul A* (*mat-pow A n*)

value *mat-pow* (*1*, *1*, *0*, *1*) *3*

thm *mat-pow.simps*

[4] Zadatak 3 iz dodatnog materijala.

lemma *mat-pow* (*1*, *1*, *0*, *1*) *n* = (*1*, *n*, *0*, *1*)
by (*induction n*, *auto simp add*: *eye-def*) — dodajemo zato što se definicije ne raspliću automatski

lemma *mat-pow* (*1*, *1*, *0*, *1*) *n* = (*1*, *n*, *0*, *1*)
proof (*induct n*)
case *0*
show *?case*
by (*simp add*: *eye-def*)
next
case (*Suc n*)
show *?case*
proof−
have *mat-pow* (*1*, *1*, *0*, *1*) (*Suc n*) = *mat-mul* (*1*, *1*, *0*, *1*) (*mat-pow* (*1*, *1*, *0*, *1*) *n*)
by *simp*
also have ... = *mat-mul* (*1*, *1*, *0*, *1*) (*1*, *n*, *0*, *1*)
using *Suc*
by *simp*
also have ... = (*1*∗*1*+*n*∗*0*, *1*∗*1*+*n*∗*1*, *0*∗*1*+*1*∗*0*, *0*∗*1*+*1*∗*1*)
by *simp*
finally show *?thesis*
by *simp*
qed
qed

6.2.2 Deljivost

Neke teoreme vezane za deljivost možemo dobiti pretragom skupa teorema:

find-theorems - *dvd* -
find-theorems - *dvd* - ∗ -
find-theorems - *dvd* - + -
find-theorems - ⟹ - *dvd* - + -

thm *nat-induct*
thm *dvd-def*
thm *dvd-add*

thm *dvd-refl*
find-theorems - ^ -
thm *power-0*
thm *power-one-right*

Zadatak 6.6. Dokazati da za svaki prirodan broj n važi: $6|n * (n + 1) * (2n + 1)$. [5]

U postavci ovog zadatka mora biti naglašeno da je u pitanju prirodan broj. U dokazima teorema sa celim brojevima, tipovi su često problem, pa se može iskoristiti nareda *using* [[*show-types*]] koja kontroliše ispis tipova promenljivih. Podrazumevano ponašanje Isabelle/HOL jeste da ne prikazuje tipove, pa korišćenje ove naredbe može da otkrije potrebu za dodavanjem informacija o tipovima promenljivih u teoremi. Nakon utvrđivanja tipova koji nedostaju, ova naredba može da se obriše.

Ako bismo prvo pokušali da formulišemo teoremu na sledeći način (namerno menjamo tvrđenje teoreme da ne bi zbunili sledgehammer kasnije sa identičnim tvrđenjem), i pokušali sa pravilom indukcije dobili bismo poruku o grešci: *Unable to figure out induct rule.* Ako sada dodamo naredbu *using* [[*show-types*]], odmah nakon tvrđenja teoreme (pre poziva indukcije), vidimo da Isabelle/HOL nije u stanju da ustanovi kog tipa su vrednost 6 i 2. Pa dodajemo sa *fixes* informaciju da je n prirodan broj. Sada ako ponovo pogledamo izlaz nakon naredbe *using* [[*show-types*]] možemo primetiti da Isabelle/HOL sada može da zaključi tip i za vrednosti 6 i 2.

```
declare [[quick-and-dirty=true]]

lemma deljivost-primer:
shows 6 dvd 2*3*n
using [[show-types]]
— proof (induction n)
  sorry

lemma deljivost-sa-6:
fixes n :: nat
shows 6 dvd n*(n+1)*(2*n+1)
— using [[show-types]]
proof (induction n)
  case 0
  then show ?case
    by simp
next
  case (Suc n)
  then show ?case
  proof −
    have Suc n * (Suc n + 1) * (2 * Suc n + 1) = (n+1)*(n+2)*(2*n+3)
```

[5] Zadatak 9 iz dodatnog materijala.

```
    by (simp add: algebra-simps)
  also have ... = n*(n+1)*(2*n+3) + 2*(n+1)*(2*n+3)
    by (simp add: algebra-simps)
  also have ... = n*(n+1)*(2*n+1) + 2*n*(n+1) + 2*(n+1)*(2*n+3)
    by (simp add: algebra-simps)
  also have ... = n*(n+1)*(2*n+1) + 2*(n+1)*(3*n+3)
    by (simp add: algebra-simps)
  also have ... = n*(n+1)*(2*n+1) + 6*(n+1)*(n+1)
    by (simp add: algebra-simps)
  finally show ?thesis
    using Suc
    — na ovakvom mestu mozemo pokusati i sledgehammer
    thm One-nat-def
— a možemo pokušati i using [[simp-trace]] by simp i dobijamo poruku: "Term does
not become smaller". Ovaj izlaz nam prikazuje šta je tačno primenjivano tokom
simplifikacije. Ako krene pogrešnim putem onda se može obrisati teorema koja u
stvari nije potrebna za ispravan dokaz. Ovde je to teorema One-nat-def: 1 = Suc
0.
    by (simp del: One-nat-def)
 qed
 qed
```

Zadatak 6.7. Dokazati da za svaki prirodan broj n važi: $3|5^n + 2^{n+1}$. [6]

```
lemma deljivost-sa-3:
fixes n :: nat
shows (3::nat) dvd 5^n + 2^(n+1)

proof (induct n)
 case 0
 thus ?case
— sledgehammer daje slično rešenje
— drugi način je da pokušamo sa automatskim dokazivačima sami, i naredba apply
simp sugeriše da je potrebno koristiti pravilo narednog tipa 3 = Suc (Suc (Suc 0))

   by (simp add: numeral-3-eq-3)
next
 case (Suc n)
— u narednom redu se koristi konvertovanje u tip nat. Konvertovanje je neophodno
kada iz konteksta nije moguće zaključiti tip. Operator stepenovanja ( ^) je polimor-
fan i primenjuje se na objekte različitih tipova. U situaciji kada ne prolazi dokaz
```

[6] Zadatak 10 iz dodatnog materijala.

koji deluje da bi trebalo da prođe automatski moguće je da tipovi nisu u redu i da se moraju precizirati.

have (*5*::*nat*) ^ *Suc n* + *2* ^ (*Suc n* + *1*) = *5* ∗ *5* ^ *n* + *2* ∗ *2* ^ (*n* + *1*)
using[[*show-types*]] — pokrenite ovu naredbu bez ::nat u prethodnom redu
by *simp*

also have ... = *3* ∗ *5* ^ *n* + *2* ∗ (*5* ^ *n* + *2* ^ (*n* + *1*))
by *simp*
finally
show *?case*
— ovde se može iskoristiti sledgehammer, ili možemo pogledati sam oblik koji nam treba i napakovati sledeći korak:
— ispis teorema koje se koriste:
thm *Suc*
thm *dvd-triv-left*
thm *dvd-mult*
thm *dvd-add*
using *dvd-add*[*OF dvd-triv-left*[*of 3*::*nat 5*^*n*] *dvd-mult*[*OF Suc*, *of 2*]]
by *simp*
qed

Objašnjenje korišćenih naredbi:

dvd-triv-left[*of*...] - `of` služi da instanciramo promenljive u lemi, navodimo ih onim redom kojim se javljaju u njoj.

dvd-add[*OF*...] nam omogućava da zadamo pretpostavke teoreme *dvd-add*, pretpostavke su u stvari dve činjenice koje dobijamo uz pomoć teorema *dvd-triv-left* i *dvd-mult*.

dvd-mult[*OF Suc*, *of 2*] - `OF` omogućava nam da zadamo pretpostavku našoj lemi, ovako istovremeno zadajemo i pretpostavku i instanciramo jednu promenljivu (sa `of`).

6.2.3 Nejednakosti sa celim brojevima

Zadatak 6.8. Dokazati da za svaki prirodan broj $n \geq 5$ važi $n^2 < 2^n$. [7]

U udžbeniku možemo da primetimo da se u dokazu ovog tvrđenja koristi pomoćno tvrđenje koje prvo moramo dokazati. Prilikom formulisanja ovih tvrđenja moramo naglasiti da je n prirodan broj, zato što to ne može da se zaključi iz konteksta a ta informacija nam je neophodna da bismo mogli u dokazima da koristimo princip matematičke indukcije. To radimo ključnom rečju *fixes*.

[7] Zadatak 11 iz dodatnog materijala.

Ključna reč *fixes* nam omogućava da navedemo tip promenljivih unapred, pre navođenja tvrđenja teoreme (naspram korišćenja *n*::*nat* negde u zapisu teoreme). Pored toga ovde nam znači da koristimo struktuirani zapis tvrđenja i ključne reči *assumes* i *shows* koje nam dozvoljavaju da imenujemo pretpostavke teoreme (ako imamo više različitih pretpostavki možemo ih razdvojiti ključnom rečju *and*) i sa *shows* da izdvojimo cilj teoreme.

Sledeći detalj koji moramo naglasiti jeste da nam je u ovom dokazu neophodno da uključimo pretpostavke pre pozivanja indukcije. To radimo korišćenjem ključnih reči *using assms* i sada je uslov $n > 2$ uključen i u princip indukcije. U ovakvim dokazima induktivni korak se sastoji iz dva dela, prvi deo je standardni (da tvrđenje važi za n), a drugi deo se sastoji od tvrđenja da sledbenik broja n zadovoljava pretpostavke koje se nalaze među *assms*.

Sada moramo obratiti pažnju na dva nivoa baze indukcije, prvi obrađuje šta se dešava sa nulom (automatski je ispunjen uslov zato što nula ne zadovoljava pretpostavke teoreme), a drugi nivo obrađuje šta se stvarno dešava sa minimalnim brojem n za koji je ispunjen preduslov našeg tvrđenja. U ovom primeru to je broj 3. Pa sada treba na neki način posebno obraditi slučaj trojke, i slučaj sledbenika.

Ovakva induktivna hipoteza je uslovna, uslov koji mora da važi je da je *Sled* $n \geq 3$, odnosno $n \geq 2$. Dakle razlikujemo slučaj kada je *Sled* $n = 3$ i slučaj kada je *Sled* $n > 3$. Slučaj kada je *Sled* $n = 3$, je stvarna baza indukcije. I moramo da izvršimo analizu slučajeva prema ova dva slučaja.

Rezonovanje po slučajevima se dobija metodom *cases* i u ovom slučaju poziva se naredbom *proof* (*cases Suc n = 3*). Dobijamo dva slučaja *case True* i *case False*. U ovom dokazu takođe moramo naglasiti da je *also have* rezonovanje pogodno za operator $\leq$ ali nije pogodan za lanac koji u sebi ima $\geq$ (kao što je navedeno u dokazu u udžbeniku), pa se vodi računa o smeru u kom izvodimo formule.

thm *power2-eq-square*

lemma *n2-2np1*:
  fixes *n*::*nat*
  assumes *n > 2*
  shows *n^2 > 2*n + 1*
  using *assms*
proof (*induction n*)
  case *0*
  thus *?case* — slučaj 0 se lako dokazuje zato što leva strana nije zadovoljena pa je i desna automatski tačna, šta god pisalo na tom mestu
  by *simp*

```
next
  case (Suc n)
  thm Suc
  show ?case
  proof (cases n = 2)
    case True
    thus ?thesis
      by simp
  next
    case False

    hence n > 2
      using Suc(2)
      by simp

    have 2 * (Suc n) + 1 < 2 * (Suc n) + 2 * n
      using ‹n > 2›
      by simp
    also have ... = 2 * n + 1 + 2 * n + 1
      by simp
    also have ... < n^2 + 2 * n + 1
      using Suc(1) ‹n > 2›
      by simp
    also have ... = (Suc n)^2
      by (simp add: power2-eq-square)
    finally
    show ?thesis
      .
  qed
qed
```

Sada prelazimio na dokaz glavnog tvrđenja koji je slične strukture. Jedina nova stvar u ovom dokazu je instanciranje promenljive iz leme uz pomoć naredbe [*of n*]. U uglastim zagradama se navodi ime promenljive.

```
lemma
  assumes n ≥ 5
  shows 2^n > n^2
  using assms
proof (induction n)
  case 0
  thus ?case
    by simp
next
```

```
  case (Suc n)
  show ?case
  proof (cases n = 4) — suštinska baza indukcije
    case True
    thus ?thesis
      by simp
  next
    case False
    hence n ≥ 5
      using Suc(2)
      by simp

    have (Suc n) ^2 = n^2 + 2*n + 1
      by (simp add: power2-eq-square algebra-simps)
    also have ... < n^2 + n^2
      using Suc(2) n2-2np1[of n] — instanciranje promenljive
      by simp
    also have ... = 2 * n^2
      by simp
    also have ... < 2 * 2^n
      using Suc(1) ‹n ≥ 5›
      by simp
    also have ... = 2^(n+1)
      by simp
    finally
    show ?thesis
      by simp
  qed
qed
```

6.3 Pravilo indukcije koje počinje pozitivnim brojem

Pravilo indukcije koje zadovoljava uslov $n \geq m$ zove se *nat-induct-at-least*: $[\![m \leq n;\ P\ m;\ \bigwedge n.\ [\![m \leq n;\ P\ n]\!] \Longrightarrow P\ (Suc\ n)]\!] \Longrightarrow P\ n$ i ovako se zapisuje u Isabelle/HOL. Zahvaljujući ovom pravilu možemo zapisati prethodne dokaze tako da više liče na dokaze iz udžbenika.

thm *nat-induct-at-least*

lemma *n2-2np1-at-least*:
fixes *n*::*nat*
assumes $n \geq 3$
shows $n\hat{}2 > 2*n + 1$
using *assms*
proof (*induction n rule*: *nat-induct-at-least*)
— bazni slučaj
case *base*
thus *?case*
by *simp*
next
case (*Suc n*)
have $2 * (Suc\ n) + 1 < 2 * (Suc\ n) + 2 * n$
using ‹$n \geq 3$›
by *simp*
also have ... $= 2 * n + 1 + 2 * n + 1$
by *simp*
also have ... $< n\hat{}2 + 2 * n + 1$
using *Suc*
by *simp*
also have ... $= (Suc\ n)\hat{}2$
by (*simp add*: *power2-eq-square*)
finally
show *?case*
.
qed

6.4 Princip jake indukcije

Postoje dva metoda: `less_induct`: $(\bigwedge x.\ (\bigwedge y.\ y < x \Longrightarrow P\ y) \Longrightarrow P\ x) \Longrightarrow P\ a$ i `nat_less_induct`: $(\bigwedge n.\ \forall\, m{<}n.\ ?P\ m \Longrightarrow ?P\ n) \Longrightarrow ?P\ ?n$. Prvi je opštiji (i radi za bilo koje potpuno uređenje), ali drugi je specifičniji i zadovoljava naše potrebe za sada.

thm *less-induct*
thm *nat-less-induct*

Zadatak 6.9. Niz fibonačijevih brojeva zadat je jednačinama: $f_1 = 1$, $f_2 = 1$, $f_n = f_{n-1} + f_{n-2}$, za $n > 2$.
Dokazati da važi nejednakost:

$$f_n \leq \varphi^{n-1}$$

za svaki prirodan broj n, gde je $\varphi = \frac{1+\sqrt{5}}{2}$. [8]

```
definition gm where
 gm = (1 + sqrt 5) / 2

fun fib :: nat ⇒ nat where
fib 0 = 0 |
fib (Suc 0) = 1 |
fib (Suc (Suc n)) = fib (Suc n) + fib n

lemma fibonaci:
shows fib (n + 1) ≤ gm ^ n
proof (induction n rule: nat-less-induct)
case (1 n)
thm 1
  show ?case
  proof (cases n = 0)
    case True
    thus ?thesis
      by simp
  next
    case False
    show ?thesis
    proof (cases n = 1)
      case True
      have fib 2 = 1
        by (simp add: numeral-2-eq-2)
        thus ?thesis
        using ‹n = 1›
        by (simp add: gm-def)
    next
      case False
      show ?thesis
      proof−
        obtain m where m = n − 2
          by auto
          have n > 1 using ‹n ≠ 0› ‹n ≠ 1›
          by simp
        have fib (m + 3) = fib (m + 2) + fib (m + 1)
          by (simp add: numeral-3-eq-3)
        also have ... ≤ gm ^ (m + 1) + gm ^ m
```

[8] Zadatak 6 iz dodatnog materijala.

— modifikator [*rule-format*] od $\forall x.Px$ pravi "P ?x" na koji onda možemo dalje da primenimo of da bi ubacili vrednost za x. Obično se koristi kada imamo pretpostavku sa univerzalnim kvantifikatorom koju hoćemo da instanciramo prilikom primene.

```
        using 1[rule-format, of m+1] 1[rule-format, of m] ‹m = n − 2› ‹n > 1›
        by force
      also have ... = gm ^ m ∗ (gm + 1)
        by (simp add: algebra-simps)
      also have ... = gm ^ m ∗ (gm ^ 2)
        by (simp add: power2-eq-square algebra-simps gm-def)
      also have ... = gm ^ (m + 2)
        by (simp add: power-add power2-eq-square)
      finally
      show ?thesis
        using ‹m = n − 2› ‹n > 1›
        by (simp add: numeral-2-eq-2 Suc-diff-Suc)
    qed
  qed
 qed
qed
```

6.5 Realni brojevi

Realni brojevi nisu uključeni u teoriju *Main*. Da bi mogli da radimo sa realnim brojevima potrebno je da uključimo *HOL.Real* a ako želimo da radimo i sa kompleksnim brojevima potrebno je da uključimo *MainComplex-Main*

Zadatak 6.10. Dokazati da važi Bernulijeva nejednakost $(1+x)^n \geq 1+nx$, za $x > -1$. [9]

U narednom dokazu, nećemo koristiti *using assms* da bi bili jednostavniji dokazi, pa ćemo se pozivati na pretpostavku samo na onim mestima u dokazu gde ćemo je eksplicitno koristiti.

```
lemma bernulijeva-nejednakost:
  fixes x::real and n::nat
  assumes x > −1
  shows (1 + x)^n ≥ 1 + n ∗ x
proof (induction n)
  case 0
  show ?case
```

[9] Zadatak 4 iz dodatnog materijala.

```
    by simp
next
  case (Suc n)
  show ?case
  proof–
    have 1 + (n + 1) * x ≤ 1 + (n + 1) * x + n * x^2
      by simp
    also have ... = (1 + x) * (1 + n * x)
      by (simp add: algebra-simps power2-eq-square)
    also have ... ≤ (1 + x) * (1 + x) ^ n
      using Suc ‹x > −1› — ovde se pozivamo na pretpostavku
      by simp
    also have ... = (1 + x) ^ (Suc n)
      by simp
    finally
    show ?thesis
      by simp
  qed
qed
```

Funkcije za kastovanje između razlicitih tipova se mogu naci narednim naredbama. Funkcije se zovu *int*, *real*, *real-of-nat*,...

```
find-consts nat ⇒ int
find-consts nat ⇒ real
find-consts real ⇒ complex
```

Primer funkcije *int*. Tip i funkcija mogu imati isto ime. Ovde int predstavlja funkciju koja slika tip *nat* u tip *int*.

```
term 3::nat
term int (3::nat)
term int 3
```

Poziv `@{term t}` štampa dobro definisan term t. Poziv funkcije `@{value t}` računa vrednost terma t i štampa rezultat.

```
value int (3::nat)
term 3::nat
value 3::nat

thm Re-complex-of-real
thm Im-complex-of-real
thm complex-of-real-def
```

Zadatak 6.11. Dokazati Moavrovu formulu: neka je dat kompleksan broj

u trigonometrijskom obliku $z = r(\cos\theta + i * \sin\theta)$. Tada se n-ti stepen broja z računa po formuli:

$$z^n = r^n(\cos n\theta + i \sin n\theta)$$

[10]

```
lemma moavrova-formula:
  fixes r φ :: real and n :: nat
  shows (r * (cos φ + i * sin φ)) ^n = r^n * (cos (n * φ) + i * sin(n * φ))
proof (induction n)
  case 0
  show ?case
    by simp
next
  case (Suc n)
  show ?case
  proof−
    have (r * (cos φ + i * sin φ)) ^ Suc n =
          (r * (cos φ + i * sin φ)) ^ n * r * (cos φ + i * sin φ)
      by simp
    also have ... = r^n * (cos (n * φ) + i * sin (n * φ)) * r * (cos φ + i * sin φ)
      using Suc
      by simp
    also have ... = r^(Suc n) * ((cos (n * φ) * cos φ − sin (n * φ) * sin φ) + i
* (sin(n * φ) * cos φ + cos (n * φ) * sin φ))
      by (simp add: algebra-simps)
      also have ... = r^(Suc n) * (cos (n * φ + φ) + i * sin (n * φ + φ))
      thm cos-add
      thm sin-add
    using cos-add[of n * φ φ, symmetric] sin-add[of n * φ φ, symmetric]
    by simp
    finally
    show ?thesis
      by (simp add: algebra-simps)
  qed
qed



end
```

[10] Zadatak 12 iz dodatnog materijala.

7

Programiranje i verifikacija

theory *Cas9-vezbe*
imports *Main*
begin

7.1 Definisanje novih tipova podataka i novih operatora

U ovom poglavlju biće prikazano na koji način možemo da definišemo novi tip podataka, operacije nad tim tipom, kao i na koji način možemo da definišemo i dokažemo svojstva tako uvedenih operacija.

7.1.1 Građenje skupa prirodnih brojeva

U ovom poglavlju ćemo videti kako možemo da izgradimo skup prirodnih brojeva od početka. Iako je ovaj skup već definisan u okviru Isabelle/HOL, detaljno će biti prikazan način za kreiranje novog rekurzivnog skupa podataka.

Skup prirodnih brojeva je određen kao najmanji skup koji sadrži nulu (0) i, za svaki broj koji već pripada tom skupu, sadrži i njegovog sledbenika. Skup prirodnih brojeva je induktivni skup, odnosno najmanji skup zatvoren za ovaj skup operacija.

Isabelle/HOL nudi definisanje novih algebarskih tipova podataka kroz konstrukciju *datatype*. Nakon toga se uvodi konstanta *nula*, koja predstavlja osnovni term od koga se polazi, i konstruktor *Sled* koji kreira nove termove (na osnovu datih elemenata ovog tipa konstruiše nove elemente ovog tipa):

datatype prirodni = nula | Sled prirodni

Sada ga Isabelle/HOL prepoznaje kao novi tip, što mozemo videti naredbom: *typ prirodni*.

Nakon ovako definisanog tipa, prirodni brojevi se sada mogu zapisati u Isabelle/HOL kao: *nula*, *Sled nula*, *Sled* (*Sled nula*). Da ne bismo u svim narednim dokazima pisali naziv konstante (nula), možemo u definiciji osnovnu konstantu zameniti nekim simbolom, npr. zadebljana nula koja se zapisuje `\<zero>`. Sada je svejedno koji ćemo zapis od ova dva koristiti.

datatype *prirodni = nula* (**0**) | *Sled prirodni*

typ *prirodni*

term 0

term *nula*
term *Sled* **0**
term *Sled* (*Sled* **0**)

Definicije: Pored konstante nula, možemo navesti i nove konstante korišćenjem definicija. Na primer konstantu jedan, dva itd., uvodimo ključnom rečju *definition*.

definition *jedan-def* :: *prirodni* **where**
jedan-def = *Sled* **0**

definition *dva-def* :: *prirodni* **where**
dva-def = *Sled jedan-def*

Sada ćemo definisati neke funkcije nad skupom prirodnih brojeva korišćenjem *primitivne rekurzije*. Primitivna rekurzija je specijalni oblik rekurzije koji direktno prati definiciju tipa podataka i slučajeve koji odgovaraju tipu.

Primitivna rekurzija: Definisati primitivnom rekurzijom funkciju sabiranja nad ovako uvedenim skupom prirodnih brojeva. Funkcija sabiranja ima dva argumenta i kao rezultat vraća vrednost njihovog zbira.

Kada funkcija koju definišemo ima dva argumenta, možemo birati da li želimo da rekurzivna definicija ide po prvom ili po drugom argumentu. U ovom primeru ćemo izabrati da bude po prvom argumentu.

Ako želimo da umesto imena funkcije *saberi* koristimo operator, njega uvodimo u zagradama između tipa i ključne reci *where*. Običan simbol + ne možemo koristiti zato što je rezervisan (isto kao i što ne koristimo običan simbol nula - 0). Koristimo simbol `\<oplus>`, broj 100 označava prioritet operatora, l (na kraju ključne reči *infixl*) označava levo asocijativni operator tj. a + b + c = (a + b) + c. Funkcionalnost je ista sa ili bez operatora, samo je notacija malo udobnija za čitanje. U narednom poglavlju biće navedeno više detalja o uvođenju novih operatora.

primrec *saberi* :: *prirodni* ⇒ *prirodni* ⇒ *prirodni* (**infixl** ⊕ *100*) **where**
0 ⊕ *y* = *y* |
(*Sled x*) ⊕ *y* = *Sled* (*x* ⊕ *y*)

Kada se definicija navede primitivnom rekurzijom, formule koje su navedene u okviru te definicije se automatski smatraju dokazanim teoremama što možemo videti pozivanjem naredne naredbe:

print-theorems

Dobijene teoreme se zovu saberi.simps i automatski su na raspolaganju simplifikatoru, što ćemo videti u narednim dokazima.

thm *saberi.simps*(*1*)
thm *saberi.simps*(*2*)

Ispis ovih funkcija u Isabelle/HOL izgleda ovako:

$\mathbf{0} \oplus y = y$

$Sled\ x \oplus y = Sled\ (x \oplus y)$

Sledećom naredbom dobijamo kao rezultat dva ali nije ispisana vrednost dva. Definicije koje navodimo nam služe da mi možemo da koristimo pojmove koje smo uveli, ali kada Isabelle/HOL dobije rezultat koji je jednak desnoj strani nekog definisanog pojma, on ga ne redukuje i ne prikazuje u obliku leve strane.

value *saberi jedan-def jedan-def* — Ispisuje se vrednost *Sled* (*Sled* **0**)

Skraćenice (eng. abbreviations): ima smisla koristiti kada se uvodi koncept koji je jednostavna varijacija već postojećeg koncepta. One se automatski raspisuju, pa nisu pogodne da se koriste nad velikim skupom već postojećih koncepta. Nisu ekvivalentne definicijama i ne mogu ih zameniti.

Umesto definicije prirodnih brojeva jedan i dva, možemo koristiti skraćenice i ključnu reč *abbreviations* kao u narednoj naredbi (simbol ekvivalencije se dobija kada otkucamo == ili \<equiv>!):

abbreviation *jedan* :: *prirodni* **where**
$jedan \equiv Sled\ \mathbf{0}$

abbreviation *dva* :: *prirodni* **where**
$dva \equiv Sled\ jedan$

Sada kada Isabelle/HOL primeti da je našao desnu stranu neke skraćenice, on će je automatski prikazati u obliku leve strane te skraćenice što se može videti u Isabelle/HOL pokretanjem naredne naredbe:

value *saberi jedan jedan* — Ispisuje se vrednost *dva*.

Još jedan primer upotrebe skraćenica je za uvođenje relacije između elemenata uređenog para, ili za uvođenje skraćenog zapisa kao što je već urađeno za negaciju jednakosti (simbol $\neq$). Naredni primer je već deo Isabelle/HOL:

```
abbreviation not_equal :: "'a \<Rightarrow> 'a \<Rightarrow> bool" (infixl "~=" 50)
where "x ~= y  \<equiv>  \<not> (x = y)"

notation (xsymbols) not_equal (infix "\<noteq>" 50)
```

Sada možemo preći na dokazivanje nekih svojstava sabiranja. Pravilo kojeg se često pridržavamo: kad god je funkcija definisana induktivno po nekom parametru, dokaz njene korektnosti se izvodi indukcijom upravo po tom parametru.

Napomena: u ovim dokazima automatski dokazivači ne pomažu zato što ne rade sa indukcijom, i to se odnosi i na *sledgehammer*, i oni uopšte ne pokušavaju da izvedu dokaze nad indukcijom. Tako da ako se u dokazu koristi inducija korisnik mora naglasiti da želi da je koristi, što ćemo mi u ovom dokazu i uraditi. Nakon što se primeni indukcija *sledgehammer* ima šanse da dokaže cilj ali problem je što ako imamo više podciljeva u dokazu, *sledgehammer* dokazuje samo prvi cilj, tako da treba birati trenutak kada se poziva.

Zadatak 7.1. Dokazati da je sabiranje asocijativna operacija.

Kada u narednom dokazu primenimo induciju, naredbom *apply* (*induction a*), vidimo da Isabelle/HOL deli dokaz na dva cilja - bazu indukcije kada je $a = 0$ i induktivni korak gde pretpostavljamo da tvrđenje važi za proizvoljno a i iz te pretpostavke dokazujemo da tvrđenje važi za *Sled a*.

thm *prirodni.induct*

Pravilo koje se implicitno primenjuje na ovom mestu je pravilo indukcije definisano samim tipom prirodnih brojeva koje se zove *prirodni.induct* i u Isabelle/HOL izgleda ovako:

$$[\![P\ \mathbf{0};\ \bigwedge x.\ P\ x \Longrightarrow P\ (\mathit{Sled}\ x)]\!] \Longrightarrow P\ \mathit{prirodni}$$

Nakon toga pokušavamo sa *apply auto* što u ovom slučaju prolazi zato što *auto* pokušava da dokaže sve preostale ciljeve i uspeva. Kada se dokaz raspisuje na ovaj način, potrebno je na kraju navesti *done* kao ranije što smo radili u dokazima sa prirodnom dedukcijom.

lemma *saberi-asoc'*:
shows $a \oplus (b \oplus c) = (a \oplus b) \oplus c$
apply (*induction a*)
apply *auto*

done

Ovaj dokaz je mogao kraće da se zapiše sa *by* (*induction a*) *auto*. Naredba *by* daje mogućnost da se navedu jedan ili dva metoda (ali ne i više od dva).

lemma *saberi-asoc*:
shows $a \oplus (b \oplus c) = (a \oplus b) \oplus c$
by (*induction a*) *auto*

Zadatak 7.2. Dokazati da je sabiranje komutativna operacija.

Dokaz svakako kreće primenom indukcije *apply* (*induction a*) na prvi parametar, nakon čega se dobijaju dva cilja. Nakon toga naredbom *apply auto* pokušavamo da dokažemo oba cilja istovremeno, ali ni jedan cilj ne uspeva.

Napomena: u Isabelle/HOL prekopirati naredni kod i videćete izlaz.

declare [[*quick-and-dirty=true*]]

```
lemma saberi_kom:
  shows "a \<oplus> b = b \<oplus> a"
  apply (induction a)
   apply auto
  sorry
```

Pogledajmo zašto ne uspeva prvi cilj: leva strana se uprostila na b (što sledi iz definicije sabiranja - sve te jednakosti se smatraju automatski delom procesa simplifikacije i biće vidljive Isabelle/HOL), ali sa desne strane imamo b+0 ali pošto ne znamo ništa o sabiranju nule sa desne strane vidimo da je ovde neophodno da definišemo novu lemu kojom ćemo dokazati ovo tvrđenje b + 0 = b.

Lema *saberi-nula-desno* se dodaje ispred leme *saberi-kom* i njen dokaz automatski prolazi, ali sada ako u originalnom dokazu samo kažemo *apply auto* ne uspevamo da se oslobodimo prvog cilja pošto ne zna koju lemu treba da iskoristi, pa možemo da mu damo tu informaciju i da na taj način dokažemo prvi podcilj.

```
lemma saberi_nula_desno:
  shows "a \<oplus> \<zero> = a"
  by (induction a) auto

lemma saberi_kom:
```

```
  shows "a \<oplus> b = b \<oplus> a"
  apply (induction a)
  apply (auto simp add: saberi_nula_desno)
  sorry
```

Alternativa naredbi *apply* (*auto simp add*: *saberi-nula-desno*) jeste da direktno naglasimo Isabelle/HOL da određenu lemu želimo da uvek koristimo sa simplifikatorom. To radimo ključnom rečju [*simp*] koja se navodi nakon imena leme. Sada će i sama naredba *apply auto* uspeti da pronađe odgovarajuću lemu.

```
lemma saberi_nula_desno[simp]:
  shows "a \<oplus> \<zero> = a"
  by (induction a) auto

lemma saberi_kom:
  shows "a \<oplus> b = b \<oplus> a"
  apply (induction a)
   apply auto
  sorry
```

Postoje dodatne naredbe za ubacivanje i izbacivanje lema iz simplifikatora: ako želimo neku lemu koja je bila deklarisana kao simp, da izbacimo iz tog skupa koristimo ključnu reč *declare* i naredbu *simp del*:

declare saberi-nula-desno [*simp del*]

A ako želimo da lemu vratimo (dodamo) u simplifikator koristimo naredbu:.

declare saberi-nula-desno [*simp*]

Sada se bavimo drugim podciljem i vidimo da u drugom podcilju sa desne strane imamo nešto oblika $b + Sled\ a$, a to ne znamo još uvek na koji način treba da transformišemo i još jednom dodajemo novu pomoćnu lemu *saberi-Sled-desno*. Prilikom navođenja ove leme, odmah dodajemo opciju [*simp*] nakon njenog imena. Njen dokaz prolazi automatski i vraćamo se na dokaz početne leme koji takođe sada prolazi automatski.

Kompletan skup lema i njihovih dokaza izgleda ovako:

— 1.2
lemma *saberi-nula-desno*[*simp*]:
  shows $a \oplus \mathbf{0} = a$
  by (*induction a*) *auto* — sada se vraćamo na 1.3

— 1.4.
lemma *saberi-Sled-desno*[*simp*]:
shows $a \oplus Sled\ b = Sled\ (a \oplus b)$
by (*induction a*) *auto*

— 1.0 — odavde krećemo
lemma *saberi-kom*:
shows $a \oplus b = b \oplus a$
apply (*induction a*)
apply *auto*
done

Sada ovaj dokaz može da se skrati na *by* (*induction a*) *auto*.

lemma *saberi-kom′*:
shows $a \oplus b = b \oplus a$
by (*induction a*) *auto*

Napomena: voditi računa o korišćenju [*simp*] prilikom dokazivanja teorema. U nekim situacijama (kao u prethodnom primeru) nam to koristi, ali to neće važiti uvek. Na primer, za komutativnost ne stavljamo [*simp*] da ne bismo napravili beskonačnu petlju u simplifikatoru. Opšte pravilo kojeg se treba držati je da se leme ubacuju u simplifikator ako je desna strana teoreme na neki način jednostavnija od leve strane (što važi u teoremama iz prethodnog primera koje smo ubacivali u simplifikator).

Prikazaćemo dokaz komutativnosti u Isar-u u etapama. Prvi korak dokaza biće svakako indukcija i Isabelle/HOL odmah nudi opštu školjku dokaza koju možemo iskopirati (klikom na taj dokaz u donjem delu ekrana).

lemma *saberi-kom1*:
shows $a \oplus b = b \oplus a$
proof (*induction a*)
case *nula* — baza indukcije
then show *?case* **sorry**
next
case (*Sled a*) — induktivni korak
then show *?case* **sorry**
qed

— kada raspišemo prvi slučaj dobijamo:

lemma *saberi-kom2*:
shows $a \oplus b = b \oplus a$
proof (*induction a*)

```
  case nula — baza indukcije
  have 0 ⊕ b = b  by simp
  also have b = b ⊕ 0 by simp
  finally show ?case .  — also... finally nam daje tranzitivnost
— . je skraceno za assumption, moze i by simp
next
  case (Sled a)  — induktivni korak
  then show ?case sorry
qed

— a možemo biti i precizniji i umesto by simp navesi konretno koje teoreme koristimo:

lemma saberi-kom3:
  shows a ⊕ b = b ⊕ a
proof (induction a)
  case nula — baza indukcije
  have 0 ⊕ b = b
— by (simp only: saberi.simps(1))
    by (rule saberi.simps(1)) — ili ovako bez ikakve automatizacije
  also have b = b ⊕ 0
    — by (simp only: saberi-nula-desno)
— by (rule saberi-nula-desno) ne prolazi zato sto nama treba b = b + 0, a lema
tvrdi da je a + 0 = a, pa moramo da obrnemo jednakost atributom symmetric
    thm saberi-nula-desno
    thm saberi-nula-desno[symmetric]
    by (rule saberi-nula-desno[symmetric])
  finally show ?case .
next
  case (Sled a)  — induktivni korak
  then show ?case sorry
qed


— sada prelazimo na induktivni korak

lemma saberi-kom4:
  shows a ⊕ b = b ⊕ a
proof (induction a)
  case nula — baza indukcije
  have 0 ⊕ b = b
    by (rule saberi.simps(1))
  also have b = b ⊕ 0
```

```
    by (rule saberi-nula-desno[symmetric])
  finally show ?case .
next
  case (Sled a)  — induktivna hipoteza je a + b = b + a, i treba da dokazemo da
je Sled a + b = b + Sled a
  thm Sled
  have Sled a ⊕ b = Sled (a ⊕ b) — na osnovu definicije sabiranja
    by (rule saberi.simps(2))
  also have ... = Sled (b ⊕ a) — posledica induktivne hipoteze koja na ovom
mestu ima ime Sled pa pozivanje na induktivnu hipotezu se zapisuje ovako
— ili by (subst Sed, rule refl)
    using Sled by simp
  also have ... = b ⊕ Sled a
    thm saberi-Sled-desno — nama treba simetrican oblik
    thm saberi-Sled-desno[symmetric]
    by (rule saberi-Sled-desno[symmetric])
  finally show ?case .
qed
```

Napomena: ovi dokazi su previše detaljni, dovoljno nam je *by simp* u ovakvim dokazima.

Zadatak 7.3. Primitivnom rekurzijom definisati operaciju množenja.

Odmah ćemo dodati i notaciju sa većim prioritetom nego sabiranje što je imalo.

```
primrec pomnozi :: prirodni ⇒ prirodni ⇒ prirodni (infixl ⊗ 101) where
0 ⊗ y = 0 |
(Sled x) ⊗ y = x ⊗ y ⊕ y

value pomnozi dva dva
value pomnozi dva (Sled dva)

value dva ⊗ dva
```

Zadatak 7.4. Dokazati komutativnost množenja. Krećemo od 2.0.

Napomena: kada se testira naredni kod, sve između ovog teksta i leme označene sa 2.0 treba staviti pod komentar i onda deo pod deo koristiti.

```
— 2.1.
lemma pomnozi-nula-desno:
  shows a ⊗ 0 = 0
```

by (*induction a*) *auto*

— 2.2
lemma *pomnozi-Sled-desno*:
shows $a \otimes (Sled\ b) = a \oplus a \otimes b$ — pošto smo vec dokazali komutativnost sabiranja sada je svejedno kako ćemo zapisati
apply (*induction a*)
apply *auto* — uspeo je prvi cilj ali drugi cilj ne, ako pogledamo drugi cilj vidimo da je razlika u zagradama, pošto je sabiranje levo asocijativno zagrade koje se ne vide stoje sa leve strane — pa nam treba lema koja dokazuje asocijativnost sabiranja
apply (*auto simp add*: *saberi-asoc*)
done

— ili kraće zapisano:
lemma *pomnozi-Sled-desno′*:
shows $a \otimes (Sled\ b) = a \oplus a \otimes b$
by (*induction a*) (*auto simp add*: *saberi-asoc*)

— 2.0 — odavde krećemo
lemma *pomnozi-kom*:
shows $a \otimes b = b \otimes a$
apply (*induction a*)
apply (*auto simp add*: *pomnozi-nula-desno*)
— ne prolazi, slično kao za sabiranje zato što ne znamo čemu je jednako b + 0; dodajemo lemu 2.1
— posle dodavanja 2.1 - sada prolazi prvi cilj
— drugi cilj je transformisan sa leve strane ali ne zna da transformiše desnu stranu pa dodajemo novu lemu 2.2
— sada se transformiše cilj i vidimo da mu nedostaje komutativnost sabiranja i dodajemo ga uz auto
apply (*auto simp add*: *saberi-kom*)
apply (*auto simp add*: *pomnozi-Sled-desno*)
done

Ovi dokazi se formiraju korak po korak, ovako kako je opisano u teksu iznad, ali kada se jednom kreira dokaz i dodaju sve potrebne leme finalni dokaz može biti kraće zapisan: *by* (*induction a*) (*auto simp add*: *saberi-kom*)

Zadatak 7.5. Dokazati da je množenje asocijativno. Krećemo od 3.0.

— 3.1 distributivnost mnozenja prema sabiranju sa desne strane
lemma *pomnozi-saberi-distrib-desno*:

shows $(a \oplus b) \otimes c = (a \otimes c) \oplus (b \otimes c)$
apply (*induction a*)
— *apply auto* i prvi cilj prolazi
— drugi cilj ne prolazi i dobijamo da nam fali komutativnost i asocijativnost sabiranja koje možemo eksplicitno dodati
apply (*auto simp add*: *pomnozi-kom saberi-asoc pomnozi-Sled-desno*)
— a mogli smo i pozvati sledgehammer i videti šta on nudi
done

— 3.0
lemma *pomnozi-asoc*:
shows $(a \otimes b) \otimes c = a \otimes (b \otimes c)$
apply (*induction a*) — dobijamo dva cilja
apply *auto*
— prvi cilj prolazi ali drugi cilj ne prolazi; i mozemo videti da nam nedostaje distributivnost mnozenja u odnosu na sabiranje pa dodajemo lemu 3.1. Nakon što je formulišemo i dokažemo potrebno je eksplicitno dodati je u metod koji se primenjuje.
apply (*auto simp add*: *pomnozi-saberi-distrib-desno*)
done

Ovakve dokaze dobijamo kada ga raspisujemo i sami pokušavamo da dokažemo, ali kada jednom znamo kako dokaz izgleda možemo ga zapisati i kraće:

lemma *pomnozi-saberi-distrib-desno′*:
shows $(a \oplus b) \otimes c = (a \otimes c) \oplus (b \otimes c)$
by (*induction a*) (*auto simp add*: *pomnozi-kom saberi-asoc pomnozi-Sled-desno*)

lemma *pomnozi-asoc′*:
shows $(a \otimes b) \otimes c = a \otimes (b \otimes c)$
by (*induction a*) (*auto simp add*: *pomnozi-saberi-distrib-desno*)

Zadatak 7.6. Definisati operaciju stepenovanja prirodnih brojeva

Zbog prirodnog načina definisanja stepenovanja, rekurzija se mora izvesti po drugom argumentu ($x^0 = 1$, $x^{(y+1)} = x^y * x$). Pošto se u rekurzivnoj definiciji javlja jedinica, a videli smo da se broj jedan može uvesti i pomoću definicije i pomoću skraćenice, u narednom kodu je potrebno da stoji jedinica uvedena pomoću skraćenice (*abbreviation*).

primrec *stepenuj* :: *prirodni* ⇒ *prirodni* ⇒ *prirodni* (**infixl** △ *102*) **where**
$x \triangle \mathbf{0} = jedan$ |
$x \triangle (Sled\ y) = x \triangle y \otimes x$

thm *stepenuj.simps*(*1*)
thm *stepenuj.simps*(*2*)

Sa funkcijom *value* proveravamo da li smo dobro definisali našu novu funkciju:

value *stepenuj dvojka dvojka*
value *stepenuj dvojka* (*Sled dvojka*)

Sada ćemo pokušati da dokažemo neke osobine koje važe za stepenovanje.

Zadatak 7.7. Dokazati da važi: $a^1 = a$.

lemma $a \triangle$ *jedan* $= a$
  by *auto*

Ova lema se dokazuje automatski, *jedan* je skraćenica za *Sled nula*, a druga jednakost se automatski izračunava iz definicije stepenovanja.

Zadatak 7.8. Dokazati da važi: $a^{(b+c)} = a^b + a^c$

Pošto je stepenovanje definisano rekurzivno po drugom parametru, treba i indukciju da sprovedemo po drugom parametru, ali sabiranje je definisano rekurzivno po prvom parametru pa ovde biramo zbog toga indukciju po b.

— 4.0
lemma *stepenuj-na-zbir*:
  shows $a \triangle (b \oplus c) = a \triangle b \otimes a \triangle c$
  apply (*induction b*)
— apply auto
— prvi slučaj prolazi automatski, ali drugi slučaj ne prolazi, vidimo da fali komutativnost množenja (a mozda i asocijativnost)
— sledgehammer
  using *pomnozi-asoc pomnozi-kom* **by** *auto*

— kraće zapisano
lemma $a \triangle (b \oplus c) = a \triangle b \otimes a \triangle c$
  using *pomnozi-asoc pomnozi-kom* **by** (*induction b*) *auto*

U ovom dokazu možemo da primetimo da dosadašnji način dodavanja informacija automatskom dokazivaču (sa naredbom *auto simp add*) ne uspeva da dokaže našu lemu. Ali dokaz prolazi kada se koristi naredba *Using*. Razlika je u tom da kada se teoreme zadaju pomoću *Using*, tokom procesa simplifikacije one se uprošćavaju i koriste u svom uprošćenom obliku, naspram naredbe *auto simp add* koji ih ne uprošćava i koristi ih u izvornom obliku. Ove sitne razlike nekada mogu uticati na uspešnost našeg dokaza pa ima smisla pokušati sa obema verzijama i videti koja prolazi.

Zadatak 7.9. Dokazati da važi: $a^{(b*c)} = (a^b)^c$. Krećemo od 5.0.

— 5.1
lemma *stepenuj-jedan*:
shows *jedan* $\triangle$ a = *jedan*
by (*induction a*) *auto*

— 5.2
lemma *stepenuj-proizvod*:
shows $(a \otimes b) \triangle c = a \triangle c \otimes b \triangle c$
apply (*induction c*)
apply *auto* — prvi cilj nestaje
— ostaje drugi cilj koji koristi komutativnost i asocijativnost množenja
— sledgehammer
by (*metis pomnozi-asoc pomnozi-kom*)

— 5.0
lemma *stepenuj-na-proizvod*:
$a \triangle (b \otimes c) = (a \triangle b) \triangle c$
apply (*induction b*)
— apply auto
— ni prvi podcilj ne prolazi, stao je kod *jedan* = *jedan* ^ *c*, pa uvodimo novu lemu 5.1
apply (*auto simp add*: *stepenuj-jedan*)
— sada nam ostaje drugi cilj, u njegovoj levoj strani imamo -^(-+-), a desno imamo (-∗-)^- tj. stepenovanje na zbir (nju imamo) i kako se stepenuje proizvod (nju moramo da dodamo) 5.2
apply (*auto simp add*: *stepenuj-proizvod stepenuj-na-zbir*)
done

7.1.2 Uvođenje novih operatora, infiksna, prefiksna i postfiksna sintaksa operatora

Operator sabiranja nad prirodnim brojevima je prikazan simbolom +. Međutim, treba imati u vidu da su mnogi infiksni simboli (npr. +, −, ∗, ≤) predefinisani u Isabelle/HOL (slično kao i u drugim programskim jezicima). Ime odgovarajuće binarne funkcije sabiranja je zapravo zadato sa zagradama oko simbola: (+). Odnosno izraz $x + y$ je samo školjka koja je interno predstavljena izrazom (+) x y.

Korišćenje predefinisanih simbola može dovesti do toga da sistem ne može sam da odredi tip promenljive. Tada se tip može eksplicitno navesti na sledeći način: $m + (n::nat)$.

Na primeru funkcije koja računa faktorijal broja prikazaćemo uvođenje novog operatora. Definicija funkcije se uvodi primitivnom rekurzijom, na uobičajeni način. Nakon definicije, pozivom funkcije *value* možemo izračunati vrednost faktorijala za broj 3.

primrec *fact* :: *nat* ⇒ *nat* **where**
fact 0 = *1* |
fact (*Suc n*) = *fact n* ∗ (*n* + *1*)

value *fact 3*

U matematičkim udžbenicima faktorijel obično zapisujemo sa znakom uzvika $n!$, pa možemo takvu sintaksu uvesi i u Isabelle/HOL na sledeći način:

primrec *factoriel* :: *nat* ⇒ *nat* ((-!) *100*) **where**
0 ! = *1* |
(*Suc n*) ! = (*n* !) ∗ (*n* + *1*)

value *factoriel 3*
value *3*!

Kada biramo simbole kojima ćemo predstaviti neki operator, moramo voditi računa da Isabelle/HOL već ima rezervisanu određenu grupu simbola koje ne bi trebalo predefinisati. Tako da treba biti oprezan. Nesrećno izabran simbol može dovesti do greške u kodu jer Isabelle/HOL neće dozvoliti njegovo predefinisanje, pa treba voditi računa sa ovakvim primenama.

Uvođenje infiksnog operatora Dodatni simbol možemo uvesti i za proizvoljni predikat sa dva ili više argumenata. Naredna funkcija implementira izmišljeni predikat. Ključna reč **infixl** naglašava da je u pitanju infiksni operator koji je levo asocijativan.

primrec *funkcija* :: *nat* ⇒ *nat* ⇒ *nat* (**infixl** ♢ *110*) **where**
0 ♢ - = *0* |
(*Suc x*) ♢ *y* = *x* ♢ *y* + *2*∗*x* + *y*

Nakon ovakve definicije izrazi *funkcija x y* i *x* ♢ *y* interno imaju isto značenje.

U opštem slučaju, za proizvoljni predikat sa dva ili više argumenta možemo uvesti infiksnu notaciju. Pri čemu je potrebno naglasiti asocijativnost operatora ključnom reči: **infixl** - leva asocijativnost, **infixr** - desna asocijativnost, **infix** - neorijentisana asocijativnost (u tom slučaju izraz oblika *x* ♢ *y* ♢ *z* nije dozvoljen već je neophodno koristiti zagrade da bi se pravilno odredio redosled primene operacija).

Broj 101 koji se javlja nakon simbola koji uvodimo, služi za precizno definisanje prioriteta operatora. U Isabelle/HOL već postoji veliki broj ugrađenih simbola sa definisanim značenjem, ovi simboli se mogu sastojati od jednog ili više karaktera (`+`, `++`, čak i `\<and>`). Neke neobične kombinacije karaktera (kao što je na primer `[+]`) mogu biti slobodne za korisničke funkcije. Prioritet operatora je ceo broj u opsegu od 0 do 1000. Najniže vrednosti (odnosno simboli najvišeg prioriteta) su rezervisane za meta-logiku. HOL sinktaksa koristi prioritete od 10 do 100: znak jednakosti (infix =) ima vrednost prioriteta 50; logički veznici ($\vee$ i $\wedge$) imaju vrednost prioriteta ispod 50; algebarski veznici ($+$ i $*$) imaju vrednost prioriteta iznad 50. Preporuka je da korisnički uvedeni simboli imaju vrednost prioriteta između 100 i 900.

Postfiksna i prefiksna notacija unarnog operatora: Ako predikat ima samo jedan argument, operator koji se koristi mora biti naveden unutar običnih zagrada (i). Ako želimo da unarni operator bude postfiksan, moramo dodati podvlaku ispred samog operatora.

```
primrec uvecaj-post :: nat ⇒ nat ((-⋆) 100) where
0 ⋆ = 1 |
(Suc x) ⋆ = Suc (x ⋆)

value uvecaj-post 3
value 3 ⋆
```

Ako ne navedemo podvlaku ispred simbola novog operatora, dobija se prefiksna notacija unarnog operatora. Ovakav simbol se u narednim izrazima mora navoditi između običnih zagrada.

```
primrec uvecaj-pref :: nat ⇒ nat ((‡) 100) where
(‡) 0 = 1 |
(‡) (Suc x) = Suc ((‡) x)

value uvecaj-pref 3
value (‡) 3
```

Postoji mogućnost dodavanja glifa našoj kombinaciji simbola (kao što je već urađeno za ugrađene Isabelle/HOL simbole). Na primer, u narednoj definiciji uvodimo novi operator `[+]`. Naredbom `notation` možemo dodeliti "lepši" ispis ako procenimo da nam treba. Nakon toga je svejedno da li ćemo koristiti u tekstu A [+] B ili $A \odot B$.

```
definition xor :: bool ⇒ bool ⇒ bool (infixl [+] 60)
  where A [+] B ≡ (A ∧ ¬ B) ∨ (¬ A ∧ B)
```

notation (*xsymbols*) *xor* (**infixl** ⊙ *60*)

value *True* [+] *False*
value *True* ⊙ *False*
value *A* [+] *B*
value *A* ⊙ *B*

7.1.3 Dodatni primeri

Dokazati sve leme sa početka ovog poglavlja u Isar-u:

Zadatak 7.10. ”a + 0 = a”

lemma *saberi-nula-desno-isar*:
  shows *a* ⊕ **0** = *a*
proof (*induction a*)
case *nula*
  show *?case*
    by (*simp only*: *saberi.simps*(*1*))
next
case (*Sled a*)
  show *?case*
  proof−
    have *Sled a* ⊕ **0** = *Sled* (*a* ⊕ **0**)
      by (*simp only*: *saberi.simps*(*2*))
    also have ... = *Sled a*
      using *Sled*
      by *simp*
    finally
    show *?thesis*
      .
  qed
qed

Zadatak 7.11. *a* ⊕ *Sled b* = *Sled* (*a* ⊕ *b*)

lemma *saberi-Sled-desno-isar*:
  shows *a* ⊕ *Sled b* = *Sled* (*a* ⊕ *b*)
proof (*induction a*)
  case *nula*
  show *?case*
    by (*simp only*: *saberi.simps*(*1*))
next

```
  case (Sled a)
  show ?case
  proof−
    have Sled a ⊕ Sled b = Sled (a ⊕ Sled b)
      by (simp only: saberi.simps(2))
    also have ... = Sled (Sled (a ⊕ b))
      using Sled
      by simp
    finally
    show ?thesis
      by (simp only: saberi.simps(2))
  qed
qed
```

Zadatak 7.12. $a \oplus (b \oplus c) = (a \oplus b) \oplus c$

```
lemma saberi-asoc-isar:
  shows a ⊕ (b ⊕ c) = (a ⊕ b) ⊕ c
proof (induction a)
  case nula
  then
  show ?case
    by (simp only:saberi.simps(1))
next
  case (Sled a)
  then show ?case
  proof −
  have (Sled a ⊕ b) ⊕ c = Sled (a ⊕ b) ⊕ c by (simp only: saberi.simps(2))
  also have ... = Sled ((a ⊕ b) ⊕ c) by (simp only: saberi.simps(2))
  also have ... = Sled (a ⊕ (b ⊕ c)) using Sled by simp
  finally show ?thesis by (simp only: saberi.simps(2))
  qed
qed
```

Zadatak 7.13. $a \oplus b = b \oplus a$

```
lemma saberi-kom-isar:
   shows a ⊕ b = b ⊕ a
proof (induction a)
  case nula
  then show ?case by (simp only: saberi.simps(1) saberi-nula-desno)
next
  case (Sled a)
```

```
  then show ?case
  proof −
    have Sled a ⊕ b = Sled(a ⊕ b) by (simp only: saberi.simps(2))
    also have ... = Sled(b ⊕ a) using Sled by simp
    also have ... = b ⊕ Sled a by (simp only: saberi-Sled-desno)
    finally show ?thesis .
  qed
qed
```

Zadatak 7.14. Definisati funkciju *duplo* koja računa dvostruku vrednost broja.

```
primrec duplo :: prirodni ⇒ prirodni where
  duplo 0 = 0 |
  duplo (Sled n) = Sled (Sled (duplo n))

value (duplo (Sled (Sled 0)))
```

Zadatak 7.15. Dokazati da je vrednost funkcije duplo jednaka sabiranju broja sa samim sobom. Napisati i dokaz uz pomoć automatskih alata i dokaz u Isar-u.

```
lemma duplo-zbir: duplo n = n ⊕ n
  apply (induction n)
  apply (auto)
done

lemma duplo-zbir-isar: duplo n = n ⊕ n
proof (induction n)
  case nula
  then show ?case by (simp only: duplo.simps(1) saberi.simps(1))
next
  case (Sled a)
  then show ?case
  proof −
    have duplo(Sled a) = Sled(Sled(duplo a)) by (simp only: duplo.simps(2))
    also have ... = Sled(Sled(a ⊕ a)) using Sled by simp
    also have ... = Sled(Sled a ⊕ a) by (simp only: saberi.simps(2))
    finally show ?thesis by (simp only: saberi-Sled-desno)
  qed
qed
```

Zadatak 7.16. Definisati funkciju koja računa zbir brojeva od 1 do n.

```
primrec zbir-do :: prirodni ⇒ prirodni where
  zbir-do 0 = 0
| zbir-do (Sled n) = (Sled n) ⊕ (zbir-do n)
```

Zadatak 7.17. Dokazati da važi $duplo(a \oplus b) = duplo\ a \oplus duplo\ b$, uz automatske alate i u Isar-u.

Napomena: Ova lema se koristi u dokazu narednog zadatka pa je zato data ovde. Inače bio bi zadat samo naredni zadatak a na studentima bi bilo da otkriju da je ova lema potrebna.

```
lemma duplo-od-zbira:
duplo (a ⊕ b) = duplo a ⊕ duplo b
  by (induction a) auto

lemma duplo-od-zbira-isar:
duplo (a ⊕ b) = duplo a ⊕ duplo b
proof (induction a)
case nula
  then show ?case by (simp only: duplo.simps(1) saberi.simps(1))
next
  case (Sled a)
  then show ?case
  proof −
   have duplo (Sled a ⊕ b) = duplo (Sled (a ⊕ b)) by (simp only: saberi.simps(2))
    also have ... = Sled (Sled (duplo (a ⊕ b))) by (simp only: duplo.simps(2))
    also have ... = Sled (Sled (duplo a ⊕ duplo b)) using Sled by simp
   also have ... = Sled (Sled (duplo a) ⊕ duplo b) by (simp only: saberi.simps(2))
   also have ... = Sled (Sled (duplo a)) ⊕ duplo b by (simp only: saberi.simps(2))
    also have ... = duplo (Sled a) ⊕ duplo b by (simp only: duplo.simps(2))
    finally show ?thesis .
  qed
qed
```

Zadatak 7.18. Dokazati da važi $duplo(zbir\text{-}do\ n) = n \otimes Sled\ n$, uz automatske alate i u Isar-u.

```
thm pomnozi.simps(1)
thm zbir-do.simps(1)
thm duplo.simps(1)

lemma suma-do: duplo(zbir-do n) = n ⊗ Sled n
  apply (induction n)
```

apply *auto*
apply (*auto simp add*: *duplo-od-zbira*)
by (*metis duplo-zbir pomnozi.simps*(*2*) *pomnozi-kom saberi-asoc saberi-kom*)

lemma *suma-do-isar*: *duplo*(*zbir-do n*) = *n* ⊗ *Sled n*
proof (*induction n*)
case *nula*
then show *?case* **by** (*simp only*: *pomnozi.simps*(*1*) *zbir-do.simps*(*1*) *duplo.simps*(*1*))
next
case (*Sled n*)
show *?case*
thm *Sled*
proof −
have *duplo* (*zbir-do* (*Sled n*)) = *duplo* ((*Sled n*) ⊕ (*zbir-do n*))
by (*simp only*: *zbir-do.simps*(*2*))
also have ... = *duplo*(*Sled n*) ⊕ *duplo*(*zbir-do n*) **by** (*simp only*: *duplo-od-zbira*)
also have ... = *duplo*(*Sled n*) ⊕ (*n* ⊗ (*Sled n*)) **using** *Sled* **by** *simp*
also have ... = *Sled n* ⊕ *Sled n* ⊕ *n* ⊗ (*Sled n*) **by** (*simp only*: *duplo-zbir*)
also have ... = *Sled n* ⊕ (*Sled n*) ⊗ (*Sled n*) **by** (*simp add*: *pomnozi-kom saberi-asoc-isar pomnozi-Sled-desno*)
also have ... = *Sled* (*Sled n*) ⊗ (*Sled n*) **by** (*simp add*: *pomnozi-kom saberi-asoc-isar pomnozi-Sled-desno*)
also have ... = (*Sled n*) ⊗ *Sled* (*Sled n*) **by** (*simp add*: *pomnozi-kom*)
finally show *?thesis* **.**
qed
qed

Sve naredne leme dokazati u Isar-u:

lemma *a* ⊗ **0** = **0 sorry**

lemma *a* ⊗ (*Sled b*) = *a* ⊗ *b* ⊕ *a* **sorry**

lemma (*a* ⊕ *b*) ⊗ *c* = *a* ⊗ *c* ⊕ *b* ⊗ *c* **sorry**

lemma (*a* ⊗ *b*) ⊗ *c* = *a* ⊗ (*b* ⊗ *c*) **sorry**

lemma *a* ⊗ *b* = *b* ⊗ *a* **sorry**

lemma *a* ⊗ (*b* ⊕ *c*) = *a* ⊗ *b* ⊕ *a* ⊗ *c* **sorry**

end

7.2 Liste

theory *Cas10-vezbe*
imports *Main*
begin

Isabelle/HOL ima ugrađeni tip *list*, čiji elementi se navode između uglastih zagrada [i]. Ako bismo naveli samo elemente [*1*,*2*,*3*] Isabelle/HOL ne bi znao da odredi tip elementa tako da je potrebno da navedemo tip za makar jedan element liste (i onda svi ostali elementi moraju da budu istog tipa). Prazna lista se označava sa []. Tip liste se može navesti i nakon navođenja elemenata liste.

term [*1*,*2*,*3*]
term [*1*::*nat*,*2*,*3*]
term [*1*,*2*,*3*] :: *nat list*
term [] :: *nat list*

Neprazna lista se može navesti i svojim prvim elementom (glavom liste) iza koga sledi rep liste. Oznaka operacije nadopisivanja elementa na početak liste je #. Zapis u sledećem redu je skraćeni zapis za (*1* :: *nat*) # (*2* # (*3* # [])).

term (*1* :: *nat*) # [*2*,*3*]

7.2.1 Korišćenje listi za zapisivanje funkcija nad prirodnim brojevima

Zadatak 7.19. Dokazati da važi: $0 + 1 + 2 + ... + (n - 1) = n * (n - 1)/2$.

Kada na raspolaganju imamo liste, možemo ovu lemu zapisati na drugi način od onog koji smo koristili u poglavlju 6.2.

U zapisu ove leme koristimo operator ..< koji kreira listu i funkciju za sumiranje elemenata liste *sum-list*.

value [*0*..<*5*] — lista brojeva strogo manjih od 5
value *sum-list* [*0*..<*5*] — suma brojeva strogo manjih od 5

Najkraći dokaz bi bio direktnim korišćenjem matematičke indukcije nad prirodnim brojevima:

lemma *sum-list* [*0*..<*n*+*1*] = *n* * (*n* + *1*) *div 2*
  by (*induction n*) *auto*

Slično kao u poglavlju 6.2 dokaz ovog tvrđenja možemo zapisati i u Isaru na sledeći način:

lemma
sum-list [0..<n] = n∗(n−1) div 2
proof (*induction n*)
case *0*
then show *?case*
by *simp*
next
case (*Suc n*)
term *Suc*
thm *Suc* — Ispis značenja ove teoreme u ovom konkretnom dokazu *sum-list [0..<n] = n ∗ (n − 1) div 2*
note *ih = this*
have *sum-list [0..<Suc n] = sum-list [0..<n] + n*
by *simp*
also have *... = (n ∗ (n − 1)) div 2 + n*
using *ih*
by *simp*
also have *(n ∗ (n − 1)) div 2 + n = n ∗ (n + 1) div 2*
by (*metis Suc-eq-plus1 atLeastLessThanSuc-atLeastAtMost calculation distinct-sum-list-conv-Sum distinct-upt gauss-sum-nat set-upt*)
finally
show *?case*
by *simp*
qed

Zbir kvadrata

Zadatak 7.20. Dokazati da važi da je $1 + 2^2 + ... + n^2 = n * (n + 1) * (2 * n + 1) div 6$. [1]

Prvo je potrebno definisati funkciju koja opisuje levu stranu ovog izraza. Postoji više načina da definišemo funkciju koja računa zbir kvadrata od 1 do n. Prvi način je da koristimo primitivnu rekurziju, drugi način je da koristimo bibliotečke funkcije za rad sa listama.

Primitivna rekurzija: Prvo definišemo primitivnom rekurzijom funkciju koja izračunava zbir kvadrata:

primrec *zbir-kvadrata* :: *nat* ⇒ *nat* **where**
zbir-kvadrata 0 = 0 |
zbir-kvadrata (Suc n) = zbir-kvadrata n + (Suc n) ^ 2

[1] Zadatak 8 iz dodatnog materijala.

value *zbir-kvadrata 1*
value *zbir-kvadrata 2*
value *zbir-kvadrata 3*

Ugrađene funkcije za rad sa listama: Korišćenjem ugrađene podrške za liste brojeva, možemo slično kao na prošlom času zapisati izraz za zbir kvadrata. Listu elemenata od 0 do 4, odnosno njihov zbir, možemo dobiti narednim dvema naredbama:

value [*0..<5*]
value *sum-list* [*0..<5*]

Možemo umesto 0 napisati 1 i dobiti upravo ono što nam treba u postavci ovog zadatka, listu brojeva od 1 do n, odnosno zbir brojeva od 1 do n (iako je u ovom slučaju svejedno jer 0 ne utiče na vrednost izraza):

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

Međutim, nama treba zbir kvadrata pa uvodimo elemente *funkcionalnog programiranja*. Isabelle/HOL je funkcionalni programski jezik i postoji funkcional višeg reda *map* koji datu funkciju primenjuje na svaki element liste. Prvo ćemo ključnom rečju *definition* definisati novu funkciju koja kvadrira vrednost svog argumenta:

definition *kvadrat* :: *nat* ⇒ *nat* **where**
kvadrat x = *x* ∗ *x*

Naredna naredba prikazuje korišćenje funkcionala *map* i primenjuje funkciju kvadriranja na elemente liste. Rezultat te naredbe je nova lista čiji elementi su jednaki kvadratima elemenata polazne liste. Funkcija *sum-list* sumira elemente tako dobijene liste.

value *map kvadrat* [*1..<5*]
value *sum-list* (*map kvadrat* [*1..<5*])

Isti efekat se može dobiti i bez definicije, ali sada primenjujemo *anonimnu funkciju* definisanu pomoću lambda (λ) izraza:

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

Lambda izraz predstavlja anonimnu funkciju čiji se parametar navodi odmah iza λ, a nakon tačke se nalazi telo funkcije. Sada funkciju koja izračunava zbir kvadrata možemo definisati i direktno bez primitivne rekurzije:

definition *zbir-kvadrata′* :: *nat* ⇒ *nat* **where**
zbir-kvadrata′ n = *sum-list* (*map* (λ *x*. *x* ^ *2*) [*1..<Suc n*])

U Isabelle/HOL postoji skraćeni zapis (blizak matematičkom zapisu na koji smo navikli). $\sum$ se zapisuje kao `\<Sum>`, $\leftarrow$ se zapisuje kao `\<leftarrow>`. Leva strana (pre tačke) navodi opseg vrednosti za x, a desna strana (posle tačke) navodi na koji način se te vrednosti transformišu:

value $\sum\ x \leftarrow [1..<5].\ x\ \hat{}\ 2$

Ako bismo ovu istu funkciju prosledili kao parametar funkciji *term* možemo videti da u pozadini sistem ipak koristi raspisani oblik koji smo malopre pokazali *sum-list* (*map power2* $[1..<5]$). Jedina razlika je što sistem koristi već ugrađenu funkciju *power2* koja izračunava kvadrat broja.

term $\sum\ x \leftarrow [1..<5].\ x\ \hat{}\ 2$

Zadatak 7.21. Pokazati da su ove definicije ekvivalentne, odnosno pokazati da važi: `zbir_kvadrata n = zbir_kvadrata' n`.

Prvo pokušavamo sa indukcijom, što očekivano rezultuje sa dva cilja. Nakon toga pokušavamo sa auto, ali automatski alati ne prolaze. Razlog za to je što funkcije definisane primitivnom rekurzijom automatski ulaze u simplifikator i njihova pravila se automatski primenjuju, ali funkcije definisane pomoću ključne reči *definition* nisu deo simplifikatora i moramo ih eksplicitno dodati.

thm *zbir-kvadrata.simps* — ove teoreme su automatski ukljucene u proces simplifikacije
thm *zbir-kvadrata'-def* — ova teorema nije automatski ukljucena u proces simplifikacije

lemma *zbir-kvadrata n = zbir-kvadrata' n*
apply (*induction n*)
— apply auto
apply (*auto simp add*: *zbir-kvadrata'-def*)
done
— ili skraćeno *by* (*induction n*) (*auto simp add*: *zbir-kvadrata'-def*)

Drugi način na koji možemo zapisati ovu lemu je bez uvođenja definicije, i sada nema potrebe za dodavanjem dodatne leme (zato što smo već raspisali definiciju sami):

lemma *zbir-kvadrata n* = $(\sum\ x \leftarrow [1..<Suc\ n].\ x\ \hat{}\ 2)$
by (*induction n*) *auto*

Zadatak 7.22. Dokazati da važi: $1+2^2+...+n^2 = n*(n+1)*(2*n+1)div6$

Ovaj dokaz neće proci korišćenjem samo automatskih metoda već moramo da simuliramo dokaz koji je naveden u matematičkom udžbeniku.

Pogledajmo prvo kako izgleda princip indukcije za prirodne brojeve. Postoji ugrađena teorema indukcije koja se zove *nat.induct*, $[\![P\ 0; \bigwedge nat.\ P\ nat \Longrightarrow P\ (Suc\ nat)]\!] \Longrightarrow P\ nat$, koja je već dokazana u sistemu i deo je biblioteke teorema koje mi možemo koristiti:

thm *nat.induct*

I po njoj vidimo: da bismo dokazali da svojstvo P važi za proizvoljan prirodan broj, dovoljno je da dokažemo da to svojstvo P važi za nulu, i kada fiksiramo proizvoljni prirodni broj, ako pretpostavimo da upravo za taj broj važi svojstvo P i iz toga uspemo da izvedemo da i njegov sledbenik ima svojstvo P.

lemma *zbir-kvadrata n = (n * (n + 1) * (2*n + 1)) div 6*
proof (*induction n*)
  case *0*
  then show *?case*
  by *auto*
next
  case (*Suc n*)
  have *zbir-kvadrata (Suc n) = zbir-kvadrata n + (Suc n) ^ 2*
    by *simp* — na osnovu *zbir-kvadrata.simps*
  also have *... = zbir-kvadrata n + (n+1)*(n+1)*
    by (*simp add*: *power2-eq-square*) — neophodna nam je ova lema $a^2 = a * a$
  also have *... = (n * (n + 1) * (2*n + 1)) div 6 + (n+1)*(n+1)*
    using *Suc*
    by *simp* — na osnovu induktivne hipoteze
  also have *... = (n * (n + 1) * (2*n + 1) + 6 * (n+1) * (n+1)) div 6*
     by (*simp add*: *algebra-simps*) — za algebarske manipulacije koristimo skup teorema
  also have *... = ((n + 1) * (n + 2) * (2*n + 3)) div 6*
    by (*simp add*: *algebra-simps*)
  finally
  show *?case*
    by (*simp add*: *algebra-simps*)
qed

U ovim dokazima možemo korisititi ili samo *simp* ili *auto* koji je malo moćniji. On svakako prvo poziva *simp* ali ima još neke dodatne mogućnosti koje nekada mogu biti korisne.

7.2.2 Predefinisan tip listi

Kako je tip listi induktivno definisan tip, nalik prirodnim brojevima, u ovom poglavlju ćemo pokazati kako možemo definisati novi tip podataka koji odgovara listama. Koristićemo dva konstruktora: *Prazna* koji će određivati praznu listu, i *Dodaj* koji će dodavati element na početak liste. Sledeći korak je da se pažljivo formulišu funkcije nad tip tipom i dokažu sva svojstva koja te funkcije treba da zadovoljavaju.

Uvođenje novog algebarskog tipa podataka za definisanje listi čiji su elementi prirodni brojevi se izvodi na sledeći način. Svi elementi liste moraju biti istog tipa.

```
datatype lista = Prazna | Dodaj nat lista
term "Dodaj 1 (Dodaj 2 (Dodaj 3 Prazna))"
```

Ako bismo želeli da definišemo listu sa elementima proizvoljnog-šablonskog tipa (slično generičkim klasama u jeziku Java) a ne sa fiksiranim elementima (tipa *nat*), lista se uvodi na sledeći način. Na početku definicije se sada navodi i šablonski tip jednog elementa liste *'a*.

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

Kreiranje ovako definisane liste zahteva, isto kao i u primeru sa ugrađenim listama, da se navede tip makar jednog elementa liste.

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

Ugrađene liste u Isabelle/HOL su kreirane na isti način, sa tom razlikom što su dodate skraćenice i notacije za lakši rad sa listama: konstruktor uglaste zagrade ([]) za zapisivanje prazne liste, konstruktor taraba (#) za nadovezivanje elementa na početak liste, kao i skraćeni zapis za nadovezivanje više elemenata liste [*1*,*2*,*3*]. Ovakve skraćenice je moguće dodati i za predefinisani tip liste, na sličan način kao što je urađeno sa predefinisanim tipom prirodnih brojeva.

Napomena: Sve funkcije koje definišemo za predefinisane liste, možemo definisati i za ugrađene liste i obratno. Samim tim, njihove definicije i dokazi biće skoro identični. Zbog udobnosti zapisa, većinu lema do kraja ovog poglavlja koje se tiču svojstava tih funkcija dokazivaćemo nad ugrađenim listama.

Zadatak 7.23. Definisati funkciju koja računa dužinu liste.

Kada je definisan algebarski tip podataka, funkcije za rad sa njim se definišu primitivnom rekurzijom.

Napomena: funkcije definisane nad predefinisanim tipom ćemo označiti apostrofom nakon naziva funkcije (da bismo ih razlikovali od funkcija nad ugrađenim tipom listi koje ćemo kasnije uvesti).

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

value *duzina′* (*Dodaj* (*1* :: *nat*) (*Dodaj 2* (*Dodaj 3 Prazna*)))
— izlaz: 3::nat

Zadatak 7.24. Definisati funkciju koja formira novu listu nadovezivanjem dveju datih listi jedne na drugu.

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

Zadatak 7.25. Definisati funkciju koja formira novu listu obrtanjem elemenata date liste.

U definiciji ove funkcije možemo koristiti samo funkcije koje su do sada definisane. Funkcija nadovezi' kao argumente prima dve liste, pa dodavanje elementa x na kraj liste koja se dobija rekurzivnim pozivom (*obrni′ xs*), se izvršava tako što se prvo formira lista koja sadrži samo jedan element x dodavanjem elementa x na praznu listu (*Dodaj x Prazna*).

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

term *Dodaj* (*1* :: *nat*) (*Dodaj 2* (*Dodaj 3 Prazna*))
— "Dodaj 1 (Dodaj 2 (Dodaj 3 Prazna))" :: "nat lista"
value *obrni′* (*Dodaj* (*1* :: *nat*) (*Dodaj 2* (*Dodaj 3 Prazna*)))
— "Dodaj 3 (Dodaj 2 (Dodaj 1 Prazna))" :: "nat lista"

7.2.3 Definisanje funkcija nad ugrađenim tipom listi

Na sličan način kao što smo definisali funkcije nad predefinisanim tipom listi, možemo definisati nove funkcije i nad ugrađenim tipom listi (skoro identično). Konstruktori se zovu *Nil* - prazna lista i *Cons* - dodavanje elementa na početak liste. Svaka lista je onda oblika: *Nil*, *Cons x Nil*, *Cons y* (*Cons x Nil*), itd.

Napomena: Većina svojstava će biti dokazana i automatski i u isar-u. Tamo gde nedostaje isar dokaz možete ga sami raspisati za vežbu.

Zadatak 7.26. Definisati funkciju koja računa dužinu liste i dokazati da je definisana funkcija ista kao ugrađena funkcija.

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

value *duzina* ((*1*::*nat*) # (*2* # (*3* # [])))
value *duzina* [*1*::*nat*, *2*, *3*]

Ugrađena funkcija koja računa dužinu liste naziva se *length*.

value *length* [*1*::*nat*, *2*, *3*]

Dokaz standardno ide indukcijom, kao i za sve tipove i funkcije koje su definisane primitivnom rekurzijom i pokušavamo prvo automatski, a nakon toga raspisujemo dokaz u Isar-u.

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

Dokaz u Isar-u koji počinje indukcijom se suštinski izvršava po dužini liste, iako je dužina liste unapred nepoznata. Dokaz se automatski raspisuje na dva slučaja *Nil* i *Cons*. Prvo razmatramo bazni slučaj prazne liste i nakon toga listu koja ima makar jedan element. Da bismo dokazali da neko svojstvo "P" važi za sve liste potrebno je da dokažemo:

1. bazni slučaj "P Nil" i
2. korak indukcije "P (Cons x xs)" pod pretpostavkom da važi "P xs", za neke proizvoljne ali fiksirane x i xs.

lemma *duzina-length-isar*:
  shows *duzina xs* = *length xs*
proof (*induction xs*)
  case *Nil*
  then show *?case*
    by *simp* — prolazi automatski na osnovu definicije
next
  case (*Cons a xs*) — induktivna hipoteza pretpostavlja da tvrđenje važi za xs, i pokušavamo da dokažemo da važi za (*a* # *xs*)

```
  have duzina (a # xs) = 1 + duzina xs — na osnovu definicije funkcije duzina
    by simp
  also have ... = 1 + length xs — na osnovu induktivne hipoteze
    using Cons by simp
  also have ... = length (a # xs) — na osnovu bibliotečke definicije length
    by simp
  finally show ?case
    .
qed
```

Zadatak 7.27. Definisati funkciju *prebroj* koja prebrojava koliko puta se traženi element nalazi u listi i dokazati da važi da je broj pojavljivanja elementa u listi manji ili jednak od dužine cele liste.

U ovoj definiciji se koristi *if−then−else* konstrukt. On se navodi u okviru običnih zagrada, i unutar samog izraza zagrade nisu neophodne.

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

lemma prebroj-duzina:
  shows prebroj xs x ≤ duzina xs
  by (induction xs) auto

lemma prebroj-duzina-isar:
  shows prebroj xs x ≤ duzina xs
proof (induction xs)
  case Nil
  then show ?case by simp
next
  case (Cons a xs)
  show ?case
  proof (cases x = a) — razmatramo dokaz po slučajevima
    case True
    then have prebroj (a # xs) x = 1 + prebroj xs x
      by simp — definicija prebroj
    also have ... ≤ 1 + duzina xs
      using Cons
      by simp — induktivna hipoteza
    also have ... ≤ duzina (a # xs)
      by simp — definicija duzine
    finally show ?thesis
      .
```

```
  next
    case False
    then have prebroj (a # xs) x = prebroj xs x
      by simp — definicija prebroj
    also have ... ≤ duzina xs
      using Cons
      by simp — induktivna hipoteza
    also have ... ≤ duzina (a # xs)
      by simp — definicija duzine
    finally show ?thesis
      .
  qed
qed
```

Zadatak 7.28. Dokazati da je funkcija *prebroj* ekvivalentna ugrađenoj funkciji koja broji koliko puta se element nalazi u listi *count-list*.

```
value count-list [1::nat, 2, 3, 1] 1

lemma prebroj-count-list:
  shows prebroj xs x = count-list xs x
  by (induction xs) auto
```

Zadatak 7.29. Definisati funkciju koja proverava da li data lista sadrži traženi element. Pokazati da odgovara ugrađenom mehanizmu koji je definisan u Isabelle/HOL.

Definisaćemo je primitivnom rekurzijom na sledeći način.

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

Umesto uobičajenog znaka jednakosti koji smo do sada koristili stavićemo simbol $\longleftrightarrow$. Kada radimo nad tipom *bool* između ova dva simbola (jednakost nad tipom bool i logicka ekvivalencija) nema razlike osim u prioritetu operatora. Operator logičke ekvivalencije je operator najmanjeg prioriteta pa ne zahteva zagrade sa desne strane druge jednakosti.

```
value sadrzi [1::nat, 2, 3, 5] 3
value sadrzi [1::nat, 2, 3, 5] 4
```

Ne postoji ugrađena funkcija koja proverava da li data lista sadrži traženi element. U Isabelle/HOL ova provera se vrši tako što se napravi skup

korišćenjem funkcije *set* (koja za datu listu vraća skup njenih elemenata), a onda se proveri pripadnost skupu:

value *5* ∈ *set* [*1*::*nat*, *2*, *3*, *5*]

Sada ćemo pokazati da funkcija koju smo mi definisali odgovara ovom mehanizmu.

lemma *sadrzi-in-set*:
shows *sadrzi xs a* ⟷ *a* ∈ *set xs*
by (*induction xs*) *auto*

Zadatak 7.30. Definisati funkciju koja vraća skup elemenata liste i pokazati da je ekvivalentna ugrađenoj funkciji set.

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

lemma *skup-set*:
shows *skup xs* = *set xs*
by (*induction xs*) *auto*

Zadatak 7.31. Definisati funkciju nadovezivanja dve liste i pokazati da je ekvivalentna ugrađenoj funkciji *append* koja postoji u Isabelle/HOL.

Biramo da primitivna rekurzija ide po prvom argumentu.

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

value *nadovezi* [*1*::*nat*, *2*, *3*] [*4*, *5*, *6*]

Funkcija *append* se zapisuje uz pomoć oznake @.

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

Uopšteno gledano, nema smisla definisati funkcije koje već postoje u biblioteci. Bolje je koristiti bibliotečke funkcije jer postoji već dosta lema dokazanih o njima.

find-theorems - @ -

Kada uvedemo naše funkcije, moramo dokazati da se dobro slažu sa već ugrađenim funkcijama.

Zadatak 7.32. Dokazati da je dužina nadovezane liste jednaka zbiru dužina originalnih listi. Prvo automatski pa onda u isar-u.

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

lemma *duzina-nadovezi-isar*:
shows *duzina* (*nadovezi xs ys*) = *duzina xs* + *duzina ys*
proof (*induction xs*)
case *Nil* — raspisaćemo korak po korak
have *duzina* (*nadovezi* [] *ys*) = *duzina ys*
by *simp* — na osnovu definicije koja je automatski uključena u simp
also have ... = *0* + *duzina ys*
by *simp*
also have ... = *duzina* [] + *duzina ys*
by *simp* — na osnovu definicije
finally show *?case*
.
next
case (*Cons a xs*) — krećemo od leve strane
have *duzina* (*nadovezi* (*a* # *xs*) *ys*) = *duzina* (*a* # *nadovezi xs ys*)
by *simp* — na osnovu definicije nadovezi
also have ... = *1* + *duzina* (*nadovezi xs ys*)
by *simp* — na osnovu definicije duzina
also have ... = *1* + *duzina xs* + *duzina ys*
using *Cons*
by *simp* — na osnovu induktivne hipoteze
also have ... = *duzina* (*a* # *xs*) + *duzina ys*
by *simp* — na osnovu definicije duzina
finally show *?case*
.
qed

Zadatak 7.33. Dokazati da je skup elemenata nadovezanih listi jednak uniji skupova elemenata originalnih listi.

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

Zadatak 7.34. Dokazati da se element nalazi u nadovezanoj listi ako i samo ako se nalazi makar u jednoj od originalnih listi.

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

Zadatak 7.35. Prvi i poslednji element liste se dobijaju funkcijama *hd* i *last*. Dokazati da važi *hd* (*rev xs*) = *last xs*

Prilikom formulisanja ove leme, neophodna je pretpostavka da je lista neprazna zato što je glava liste parcijalno definisana funkcija samo za neprazne liste.

lemma *xs* ≠ [] ⟶ *hd* (*rev xs*) = *last xs*
by (*induction xs*) *auto*

Obrtanje liste

Definisati primitivnom rekurzijom funkciju koja obrće elemente liste. Ovu funkciju definišemo pomoću funkcije *nadovezi* koja prima kao argumente dve liste, pa se prvi element liste (koji se sada dodaje na kraj nove liste) mora pretvoriti u listu (koja sadrži samo taj jedan element) [*x*].

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

value *obrni* [*1*::*nat*, *2*, *3*, *4*]

Bibliotečka funkcija koja obrće listu zove se *rev*

value *rev* [*1*::*nat*, *2*, *3*, *4*]

Zadatak 7.36. Dokazati da je definisana funkcija *obrni* ekvivalentna bibliotečkoj funkciji obrni.
Sa ovom lemom automatski dokaz ne prolazi pa odmah pišemo raspisani isar dokaz.

lemma *obrni-rev*:
shows *obrni xs* = *rev xs*
proof (*induction xs*)
case *Nil*
show *?case*
by *simp*
next
case (*Cons a xs*)
then show *?case*

— apply simp
— kada pokušamo sa simp vidimo da nam je potrebna lema koja tvrdi da su funkcije nadovezi i append jednake
by (*simp add*: *nadovezi-append*)
qed

Zadatak 7.37. Dokazati da važi da se dvostrukim obrtanjem liste dobija početna lista

Prvi korak u dokazu (1.1) je svakako primena indukcije. Nakon toga pokušavamo sa primenom automatskih metoda, ali vidimo da je u dokazu potrebna dodatna lema koju formulišemo i pokusavamo da dokažemo (1.2).
Pratite brojeve lema u dokazu. Dokaz se ne čita linearno odozgo na dole, već obratno.

— 1.4 dokaz ove leme prolazi automatski i vraćamo se na dokaz leme *obrni-nadovezi* u kom sada dodajemo ovu lemu (nakon čega i taj dokaz prolazi)
lemma *nadovezi-desno-prazno*:
nadovezi xs [] = *xs*
apply (*induction xs*)
apply *auto*
done

— 1.3 - dodajemo asocijativnost operacije nadoveži, čiji dokaz prolazi automatski. Sada se vraćamo na dokaz leme *obrni-nadovezi* i posmatramo šta se dešava sa automatskim dokazivačem kada mu damo na raspolaganje i ovu novu lemu
lemma *nadovezi-asoc*:
shows *nadovezi xs* (*nadovezi ys zs*) = *nadovezi* (*nadovezi xs ys*) *zs*
apply (*induction xs*)
apply *auto*
done

— 1.2 levu stranu leme dobijamo iz stanja u kom se automatski dokazivač nije snašao, a desnu stranu sami formulišemo u skladu sa konkretnim funkcijama koje koristimo
lemma *obrni-nadovezi*:
shows *obrni* (*nadovezi xs ys*) = *nadovezi* (*obrni ys*) (*obrni xs*)
apply (*induction xs*)
— apply auto
— kada pokušamo sa auto vidimo da dokazivaču nedostaje informacija o tome da li je operacija nadovezi asocijativna, pa formulišemo i dodajemo još jednu novu lemu 1.3
apply (*auto simp add*: *nadovezi-asoc*)

— sada vidimo da je dokazivač stao na mestu kada funkcija nadovezi ima praznu listu kao svoj drugi argument. Posto je funkcija nadovezi definisana rekurzivno po prvom argumentu, dokazivač sam ne može da se snađe sa ovom situacijom pa dodajemo novu lemu 1.4
apply (*auto simp add*: *nadovezi-desno-prazno*)
— sada se vraćamo na dokaz leme *obrni-obrni*, koji nakon svih ovih dodatih lema prolazi automatski
done

— Odavde krećemo: 1.0
lemma *obrni-obrni*:
shows *obrni* (*obrni xs*) = *xs*
apply (*induction xs*) — 1.1
— apply auto
— vidimo da dobijamo nešto oblika *obrni* (*nadovezi* - -) pa formulisemo i dodajemo novu lemu *obrni-nadovezi* 1.2
apply (*auto simp add*: *obrni-nadovezi*)
done

Napomena: Kada bismo obrisali sav nepotreban tekst korišćen za potrebe objašnjavanja dobili bismo naredni blok dokaza:

```
lemma nadovezi_desno_prazno:
  "nadovezi xs [] = xs"
  by (induction xs) auto

lemma nadovezi_asoc:
  shows "nadovezi xs (nadovezi ys zs) = nadovezi (nadovezi xs ys) zs"
  by (induction xs) auto

lemma obrni_nadovezi:
  shows "obrni (nadovezi xs ys) = nadovezi (obrni ys) (obrni xs)"
  by (induction xs) (auto simp add: nadovezi_asoc nadovezi_desno_prazno)

lemma obrni_obrni:
  shows "obrni (obrni xs) = xs"
  by (induction xs) (auto simp add: obrni_nadovezi)
```

Alternativa korišćenju *simp add*, odnosno alternativa eksplicitnom naglašavanju u svakom koraku koju lemu koristimo, jeste da koristimo operator [*simp*] nakon imena lema koje se koriste u narednim dokazima i na taj način dobijamo sledeći blok dokaza.

Ovako nešto možete koristiti na ispitu, i kada radite u svojim teorijama ali nije preporučljivo koristiti u velikim fajlovima koji se koriste za pokazivanje

velikog skupa lema koje nisu sve obavezno iz iste grupe (kao što je ovaj materijal koji se koristi kao deo skripte).

```
lemma nadovezi_desno_prazno [simp]:
  "nadovezi xs [] = xs"
  by (induction xs) auto

lemma nadovezi_asoc [simp]:
  shows "nadovezi xs (nadovezi ys zs) = nadovezi (nadovezi xs ys) zs"
  by (induction xs) auto

lemma obrni_nadovezi [simp]:
  shows "obrni (nadovezi xs ys) = nadovezi (obrni ys) (obrni xs)"
  by (induction xs) auto

lemma obrni_obrni:
  shows "obrni (obrni xs) = xs"
  by (induction xs) auto
```

Dodavanje elementa na kraj liste i obrtanje liste dodavanjem na kraj

Definisati funkciju *snoc* koja dodaje element na kraj liste, i rekurzivnu funkciju *rev-snoc* koja uz pomoć funkcije *snoc* obrće elemente liste.

primrec *snoc*:: *'a list* ⇒ *'a* ⇒ *'a list* **where**
snoc [] *x* = (*x* # []) |
snoc (*x1* # *xs*) *x* = *x1* # (*snoc xs x*)

primrec *rev-snoc* :: *'a list* ⇒ *'a list* **where**
rev-snoc [] = [] |
rev-snoc (*x1* # *xs*) = *snoc* (*rev-snoc xs*) *x1*

Zadatak 7.38. Pokazati da dvostrukim obrtanjem liste ovom funkcijom dobijamo originalnu listu.

— 2.1 leva strana ove leme se dobija iz dokazivača (na mestu gde je zablokirao) a desnu stranu sami pokušavamo da formulisemo. Ovaj dokaz prolazi automatski.
lemma *rev-snoc-snoc*:
shows *rev-snoc* (*snoc xs y*) = *y* # *rev-snoc xs*
apply (*induction xs*)
apply *auto*
done

— Krećemo odavde 2.0
lemma *rev-snoc-rev-snoc*:
shows *rev-snoc* (*rev-snoc xs*) = *xs*
apply (*induction xs*)
— apply auto
— kada primenimo auto vidimo da dokazivač ne zna šta da radi sa kombinacijom *rev-snoc* (*snoc* - -) pa formulisemo lemu 2.1. Nakon njenog dodavanja, dokaz prolazi.
apply (*auto simp add*: *rev-snoc-snoc*)
done

Dokaz iste leme u isar-u:

lemma *rev-snoc-rev-snoc-isar*: *rev-snoc* (*rev-snoc xs*) = *xs*
proof (*induction xs*)
case *Nil*
then show *?case* **by** *simp*
next
case (*Cons x1 xs*)
have *rev-snoc* (*rev-snoc* (*x1* # *xs*)) = *rev-snoc* (*snoc* (*rev-snoc xs*) *x1*)
by (*simp only*: *rev-snoc.simps*)
also have ... = *x1* # *rev-snoc* (*rev-snoc xs*)
by (*simp only*: *rev-snoc-snoc*)
also have ... = *x1* # *xs*
using *Cons*
by *simp*
finally show *?case* **.**
qed

Obrtanje liste u linearnom vremenu

Zadatak 7.39. Definisati funkciju za obrtanje liste koja radi u linearnom vremenu i dokazati da je ekvivalentna ugrađenoj funkciji *rev*.

Ugrađena funkcija obrtanja liste *rev* (kao i funkcija obrtanja liste koju smo mi definisali) ima kvadratnu složenost po dužini liste zato što za svaki element liste ta funkcija poziva funkciju *append* koja je linearna po prvom argumentu.
Postoji mogućnost da se kreira linearna funkcija koja obrće datu listu, ali je neophodno koristiti dodatni argument u kome će se postepeno akumulirati rezultujuća lista.

fun *itrev* :: *'a list* ⇒ *'a list* ⇒ *'a list* **where**
itrev [] *ys* = *ys* |

itrev (*x* # *xs*) *ys* = *itrev xs* (*x* # *ys*)

Ova funkcija je definisana repnom rekurzijom, uzima jedan po jedan element liste sa početka i dodaje ga na početak liste u kojoj se rezultat akumulira. U slučaju kada se ova funkcija pozove sa praznom akumulacionom listom, dobićemo listu sa istim elementima ali u obrnutom redosledu što ćemo i naknadno dokazati.

Prvo moramo primetiti da dokaz te leme ne prolazi automatski. Razlog za to je što se u dokazu te leme javlja fiksirana promenljiva, tj. prazna lista, pa moramo formulisati takvu istu teoremu ali sada sa opštom promenljivom.

Problem koji se sada javlja jeste da automatski dokazivač ima potrebu da u dokazu teoreme promeni drugi argument funkcije *itrev*. Da bi to bilo moguće potrebno je dodati ključnu reč *arbitrary* ispred imena promenljive čiju promenu vrednosti želimo da dozvolimo. Sada će oba dokaza proći automatski. (notacija *arbitrary*: *vars* univerzalno kvantifikuje promenljive pre primene indukcije).

lemma *itrev-rev-append′*:
shows *itrev xs ys* = *rev xs* @ *ys*
— apply (induction xs) apply auto
— prvo pokušamo ovako i ovo ne prolazi zato što je funkcija itrev definisana tako da menja svoj drugi argument, a u indukciji je promenljiva ys fiksirana a treba nam *a* # *ys*. Neophodno je koristiti generalizaciju tako da ovo važi za svako ys. Ovakav efekat se postiže naredbom:
apply (*induction xs arbitrary*: *ys*)
apply *auto*
done

ili kraće:

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

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

Funkcionali nad listama

Tri osnovna funkcionala nad listama u funkcionalnom programiranju su *map*, *filter* i *fold*.

Funkcija map Funkcija *map f xs* proizvodi listu koja se dobija kada se funkcija *f* primeni na sve elemente liste *xs*: *map f* [*x1*,...,*xn*] = [*f x1*,...,*f xn*] i definisana je na sledeći način:

```
fun map :: "('a \<Rightarrow> 'b) \<Rightarrow> 'a list \<Rightarrow> 'b list" where
"map f Nil = Nil" |
"map f (Cons x xs) = Cons (f x ) (map f xs)"
```

Zadatak 7.40. Definisati funkciju *preslikaj* i dokazati da je ekvivalentna ugrađenoj funkciji.

term *map*
value *map* (*λ x. x ^ 2*) [*1*::*nat*, *2*, *3*, *4*]

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

lemma *preslikaj-map*:
shows *preslikaj f xs* = *map f xs*
by (*induction xs*) *auto*

Zadatak 7.41. Definisati funkciju *intersperse* koja raspoređuje dati element iza svakog elementa liste: *intersperse a* [*x1*,...,*xn*] = [*x1*,*a*,*x2*,*a*,...,*a*,*xn*,*a*] i dokazati da važi *map f* (*intersperse a xs*) = *intersperse* (*f a*) (*map f xs*)

primrec *intersperse* :: *'a* ⇒ *'a list* ⇒ *'a list* **where**
intersperse x [] = [] |
intersperse x (*x1* # *xs*) = *x1* # *x* # *intersperse x xs*

value *intersperse* (*1*::*nat*)(*2* # *3* # *4* # [])
value *intersperse* (*1*::*nat*)([])
value *intersperse* (*1*::*nat*)(*2* # [])

lemma *map f* (*intersperse x xs*) = *intersperse* (*f x*) (*map f xs*)
by (*induction xs*) *auto*

lemma *map-isar*: *map f* (*intersperse x xs*) = *intersperse* (*f x*) (*map f xs*)
proof (*induction xs*)
case *Nil*
then show *?case* **by** *simp*
next
case (*Cons x1 xs*)

have *map f (intersperse x (x1 # xs)) = map f (x1 # x # intersperse x xs)*
by *simp* — definicija intersperse
also have *... = (f x1) # (f x) # map f (intersperse x xs)*
by *simp* — definicija map
also have *... = (f x1) # (f x) # intersperse (f x) (map f xs)*
using *Cons* **by** *simp*
also have *... = intersperse (f x) ((f x1) # map f xs)*
by (*simp only*: *intersperse.simps*)
also have *... = intersperse (f x) (map f (x1 # xs))*
by *simp* — definicija map
finally show *?case* **.**
qed

Zadatak 7.42. Definisati funkciju *intersperse-inside* koja raspoređuje dati element između elemenata liste: *intersperse-inside a* [*x1*,...,*xn*] = [*x1*,*a*,*x2*,*a*,...,*a*,*xn*] i dokazati da važi *map f* (*intersperse-inside a xs*) = *intersperse-inside* (*f a*) (*map f xs*)

Ova funkcija je za nijansu komplikovanija zato što moramo da vodimo računa o specijalnom slučaju kada lista ima samo jedan element - u tom slučaju se lista ne menja primenom ove funkcije. Proveru da li lista ima samo jedan element ćemo izvršiti primenom *if−then−else* bloka u kom ćemo ispitati da li je rep liste prazan.

primrec *intersperse-inside* :: *'a* ⇒ *'a list* ⇒ *'a list* **where**
intersperse-inside a [] = [] |
intersperse-inside a (x1 # xs) = (*if xs* = [] *then* (*x1 #* []) *else* (*x1 # a #* (*intersperse-inside a xs*)))

value *intersperse-inside* (*1*::*nat*)(*2 # 3 # 4 #* [])
value *intersperse-inside* (*1*::*nat*)([])
value *intersperse-inside* (*1*::*nat*)(*2 #* [])

lemma *map f* (*intersperse-inside x xs*) = *intersperse-inside* (*f x*) (*map f xs*)
apply (*induction xs*)
apply *auto*
done

Iako smo dokazali da su *map* i *preslikaj* ekvivalentne funkcije, pogledajmo šta bi se desilo ako bismo zamenili jednu sa drugom i dobili lemu 3.0.

Kada primenimo metod auto, nakon indukcije, vidimo da je dokazivač stao kod koraka oblika `f _ = []` pa formulišemu lemu sa pretpostavkom koja odgovara tom šablonu i dodajemo narednu lemu. Pokušaćemo dokaz da

granamo po slučajevima i vidimo da automatski dokazivač uspeva da je dokaže.

```
— 3.1
lemma preslikaj-u-praznu:
  shows preslikaj f xs = [] ⟶ xs = []
apply (cases xs)
apply auto
done

— Krace zapisano:
lemma preslikaj-u-praznu': map f xs = [] ⟹ xs = []
by (cases xs) auto

— Odavde krećemo 3.0
lemma preslikaj f (intersperse-inside x xs) = intersperse-inside (f x) (preslikaj f
xs)
  apply (induction xs)
— apply auto
— vidimo da se javlja preslikaj f - = [] pa uvodimo dodatnu lemu 3.1 kada nju
dodamo dokaz prolazi do kraja
  apply (auto simp add: preslikaj-u-praznu)
done
```

Ovo se desilo zato što, kao što smo ranije rekli, ugrađeni predikati imaju veliki skup već dokazanih lema koje možemo koristiti praktično i ne znajući da one postoje. U ovom dokazu smo istakli koja lema je korišćena i u prethodnom primeru (sa *map*) iako je eksplicitno nismo videli.

Dokaz početne leme, sa map, zapisan u Isar-u:

```
lemma map-inside-isar: map f (intersperse-inside x xs) = intersperse-inside (f x)
(map f xs)
proof (induction xs)
  case Nil
  show ?case by simp
next
  case (Cons x1 xs)
  show ?case
  proof (cases xs = [])
    case True
    thus ?thesis by simp
  next
    case False
      then have map f (intersperse-inside x (x1 # xs)) = map f (x1 # x #
```

intersperse-inside x xs)
by *simp*
also have ... = *f x1* # (*map f* (*x* # *intersperse-inside x xs*)) **by** *simp*
also have ... = *f x1* # (*f x* # (*map f* (*intersperse-inside x xs*))) **by** *simp*
also have ... = *f x1* # (*f x* # (*intersperse-inside* (*f x*) (*map f xs*)))
using *Cons* **by** *simp*
also have ... = *intersperse-inside* (*f x*) (*f x1* # (*map f xs*))
using ‹*xs* ≠ []› **by** *auto*
also have ... = *intersperse-inside* (*f x*) (*map f* (*x1* # *xs*)) **by** *simp*
finally show *?thesis* **.**
qed
qed

Funkcija filter Funkcija *filter* izdvaja sve elemente liste koji zadovoljavaju traženo svojstvo. Definiše se na sledeći način:

```
primrec filter:: "('a \<Rightarrow> bool) \<Rightarrow> 'a list \<Rightarrow> 'a list" where
"filter P [] = []" |
"filter P (x # xs) = (if P x then x # filter P xs else filter P xs)"
```

term *filter*
value *filter* (λ *x*. *x* > *0*) [−*3*::*int*, *4*, *2*, *0*, *6*, −*1*, *2*]

Zadatak 7.43. Definisati funkciju *izdvoj* i dokazati da je ekvivalentna ugrađenoj funkciji.

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

lemma *izdvoj-filter*:
shows *izdvoj P xs* = *filter P xs*
by (*induction xs*) *auto*

Zadatak 7.44. Funkcija filter može biti korišćena i za izdvajanje delioca datog celog broja.

definition *delioci* :: *nat* ⇒ *nat list* **where**
delioci n = *filter* (λ *d*. *d dvd n*) [*1*..<*n*+*1*]

value *delioci 12345*

Ako bismo želeli da dokažemo korektnost ovakve pretrage, dovoljno bi bilo da pokažemo da je ovako određen skup celih brojeva upravo jednak skupu brojeva koji dele ceo broj n. Odnosno, dokazujemo da nema delioca koji su manji od 1 ili koji su veći od n (van intervala pretrage nema rešenja). Za interval pretrage automatski sledi.

```
lemma
  assumes n > 0
  shows set (delioci n) = {d. d dvd n}
  unfolding delioci-def
  using assms
  using dvd-pos-nat[of n] dvd-imp-le[of - n]
  by force
```

Familija funkcija fold Funkcije tipa *fold* obrađuju dati skup podataka (elemente liste) u određenom redosledu i kao rezultat vraćaju jedinstvenu vrednost. Kao argument im se prosleđuje funkcija koja se primenjuje, lista nad čijim elementima primenjujemo funkciju i početna vrednost. Funkcija koja se primenjuje ima dva argumenta. U opštem slučaju takva funkcija nije asocijativna, tako da redosled primene operacije može da utiče na konačni rezultat. Takođe, u opštem slučaju tipovi argumenata takve funkcije ne moraju biti isti.

Ako operaciju primenimo na prvi element liste zajedno sa rezultatom koji se dobije kada se operacija primeni na ostatak liste dobijamo desni fold (*foldr*);

ako operaciju primenimo na sve elemente liste sem poslednjeg pa je onda iskombinujemo sa poslednjim elementom dobijamo levi fold (*foldl*);

a ako operaciju prvo primenimo na prvi element liste, pa taj rezultat iskombinujemo sa drugim elementom liste, pa taj rezultat iskombinujemo sa trećim elementom liste, itd. dobijamo običan fold (*fold*).

Inicijalna vrednost koja se zadaje uz ove funkcije se koristi za kombinovanje sa prvim ili poslednjim elementom liste (u zavisnosti koji tip fold funkcije se odabere.

U slučaju funkcije sabiranja koja je asocijativna i ima isti tip svojih argumenata, nema razlike u ponašanju ovih funkcija. Rezultat sve tri naredne operacije je isti i iznosi 6.

```
term fold
value fold (+) [1, 2, 3::nat] 0
term foldl
value foldl (+) 0 [1, 2, 3::nat]
```

term *foldr*
value *foldr* (+) [*1*, *2*, *3*::*nat*] *0*

Kada ne znamo kako je definisana funkcija možemo videti kako Isabelle/HOL tretira naredne izraze:

value *fold f* [*1*::*nat*, *2*, *3*] *a* — f 3 (f 2 (f 1 a))
value *foldl f a* [*1*::*nat*, *2*, *3*] — f (f (f a 1) 2) 3
value *foldr f* [*1*::*nat*, *2*, *3*] *a* — f 1 (f 2 (f 3 a))

Što se može i videti kroz neke od ugrađenih lema vezanih za ove funkcije:

lemma *fold f* [*a*,*b*,*c*] *x* = *f c* (*f b* (*f a x*))
by *simp*

lemma *foldr f* [*a*,*b*,*c*] *x* = *f a* (*f b* (*f c x*))
by *simp*

lemma *foldl f x* [*a*,*b*,*c*] = *f* (*f* (*f x a*) *b*) *c*
by *simp*

Zadatak 7.45. Definisati odgovarajuće korisničke funkcije i dokazati da su ekvivalentne ugrađenim funkcijama.

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

lemma *agregiraj-fold*:
shows *agregiraj f xs b* = *fold f xs b*
by (*induction xs arbitrary*: *b*) *auto*

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

lemma *agregirajl-foldl*:
shows *agregirajl f b xs* = *foldl f b xs*
by (*induction xs arbitrary*: *b*) *auto*

primrec *agregirajr* :: (*'a* ⇒ *'b* ⇒ *'b*) ⇒ *'a list* ⇒ *'b* ⇒ *'b* **where**
agregirajr f [] *b* = *b* |
agregirajr f (*x* # *xs*) *b* = *f x* (*foldr f xs b*)

lemma

shows *agregirajr f xs b = foldr f xs b*
by (*induction xs*) *auto*

end

7.3 Definisanje algebarskog tipa drveta i funkcija za rad sa njima

theory *Cas11-vezbe*
imports *Main HOL−Library.Multiset*

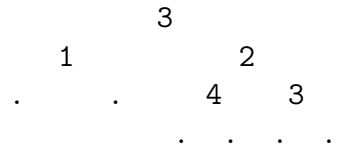
begin

Drvo je ili prazno (sto obeležavamo sa Null) ili sadrži vrednost i levo i desno poddrvo (sto obeležavamo sa Cvor l v d). Ovako definisano drvo može sadržati samo prirodne brojeve.

datatype *Drvo-nat = Null-nat | Cvor-nat Drvo-nat nat Drvo-nat*

term *Null* — prazno drvo
term *Cvor-nat* — funkcija koja od dva drveta i novog elementa kreira novo drvo
term *Cvor-nat Null 1 Null* — drvo sa jednim elementom

Primer drveta i njegovo kreiranje u Isabelle/HOL.



term *Cvor-nat* (*Cvor-nat Null 1 Null*) *3* (*Cvor-nat* (*Cvor-nat Null 4 Null*) *2* (*Cvor-nat Null 3 Null*))

Ako bismo želeli da kreiramo drvo sa elementima proizvoljnog tipa koristili bi sledeću naredbu. Sada prilikom navođenja elemenata moramo navesti i tip jednog elementa. Isto kao kod listi, dovoljno je navesti tip za jedan element pošto svi elementi drveta moraju biti istog tipa.

datatype *'a Drvo = Null | Cvor 'a Drvo 'a 'a Drvo*
term *Cvor Null* (*1*::*nat*) *Null*
term *Cvor* (*Cvor Null* (*4*::*nat*) *Null*) *2* (*Cvor Null 3 Null*)

Primeri koji slede mogu se pisati nad tipom *Drvo* kao i nad tipom *Drvo-nat*. Nema velike razlike u definicijama i funkcijama koje slede.

Zadatak 7.46. Definisati primitivnom rekurzijom funkciju koja računa zbir svih elemenata u drvetu koje sadrži samo prirodne brojeve.

Primitivna rekurzija mora da prati definiciju tipa. Posto se računa zbir, ograničili smo ovu funkciju samo na zbir drveta prirodnih brojeva.

primrec *zbir* :: *nat Drvo* ⇒ *nat* **where**
zbir Null = *0* |
zbir (*Cvor l v d*) = *zbir l* + *v* + *zbir d*

Da bismo mogli da testiramo vrednosti funkcija na konkretnom drvetu, da ne bismo iznova konstruisali drvo na koje želimo da primenimo tu funkciju, koristimo ključnu reč *definition* i uvodimo narednu definiciju radi lakšeg testiranja. Sada možemo da testiramo funkciju *zbir* primenjujuci je na *test-drvo* koje smo kreirali.

definition *test-drvo* :: *nat Drvo* **where**
test-drvo = *Cvor* (*Cvor Null 1 Null*) *3* (*Cvor* (*Cvor Null 4 Null*) *2* (*Cvor Null 3 Null*))

value *zbir test-drvo*

Zadatak 7.47. Napisati funkciju koja proverava da li drvo sadrži dati element.

Pošto ovako definisano drvo nije uređeno drvo, pretraga drveta mora da bude iscrpna i da se svaki čvor pregleda. Funkciju definišemo primitivnom rekurzijom. Ova funkcija ne bi morala da bude ograničena samo na tip prirodnih brojeva. Isto bi izgledala i za proizvoljan tip.

primrec *sadrzi* :: *nat Drvo* ⇒ *nat* ⇒ *bool* **where**
sadrzi Null - ⟷ *False* |
sadrzi (*Cvor l v d*) *x* ⟷ *sadrzi l x* ∨ *v* = *x* ∨ *sadrzi d x*

Sada testiramo funkciju *sadrzi* nad istim drvetom kao i malopre. Na isti način se može kreirati više različitih drveta pa se testirati funkcija za različitu strukturu drveta.

value *test-drvo*
value *sadrzi test-drvo 3*
value *sadrzi test-drvo 5*

Sada ćemo, analogno listama kod koje se pripadanje elementa listi proverava preko skupa elemenata koji pripadaju listi, definisati funkciju koja određuje skup svih elemenata u drvetu.

Zadatak 7.48. Definisati funkciju *skup* koja određuje skup elemenata koji pripadaju drvetu. Dokazati da je dobro definisana u odnosu na funkcije *sadrzi* i *pripadnost elementa skupu*.

Kada ne navedemo tip funkcije prilikom kreiranja, Isabelle/HOL pretpostavlja najopštiji tip tako da će ova funkcija moći da se primeni na drvo proizvoljnog

tipa (za razliku od prethodnih, gde bismo morali da promenimo tip drveta u samoj definiciji).

primrec *skup* **where**
skup Null = {} |
skup (*Cvor l v d*) = *skup l* ∪ {*v*} ∪ *skup d*

value *skup test-drvo* — testiramo funkciju skup

Veza između pretrage i skupa elemenata se formuliše narednom lemom. Ovaj dokaz prolazi automatski primenom indukcije.

lemma *pripada-skup-sadrzi*:
shows *a* ∈ *skup drvo* ⟷ *sadrzi drvo a*
by (*induction drvo*) *auto*

Raspisaćemo i dokaz u isar-u uz pomoć indukcije. Razmatramo dva slučaja, prvo slučaj praznog drveta pa onda slučaj nepraznog drveta. Slučaj praznog drveta ima obe strane koje su netačne pa su samim tim i ekvivalentne (*sadrzi Null x* = (*x* ∈ *skup Null*)). A induktivni korak ima dve induktivne hipoteze, zato što definicija drveta ima dva rekurzivna poziva, pa induktivne hipoteze odgovaraju pretpostavkama za levo, odnosno za desno poddrvo.

lemma *pripada-skup-sadrzi-isar*:
shows *a* ∈ *skup drvo* ⟷ *sadrzi drvo a*
proof (*induction drvo*)
case *Null*
show *?case* **by** *simp* — obe strane su netačne pa su ekvivalentne
next
case (*Cvor l x d*) — menjamo imena drvo1 i drvo2 sa l i d, zbog čitljivosti
then have *a* ∈ *skup* (*Cvor l x d*) ⟷ *a* ∈ *skup l* ∪ {*x*} ∪ *skup d*
by *simp* — na osnovu definicije funkcije skup
also have ... ⟷ *a* ∈ *skup l* ∨ *a* = *x* ∨ *a* ∈ *skup d*
by *auto* — na osnovu osobine unije
also have ... ⟷ *sadrzi l a* ∨ *a* = *x* ∨ *sadrzi d a*
using *Cvor*
by *simp* — na osnovu induktivne hipoteze
also have ... ⟷ *sadrzi* (*Cvor l x d*) *a*
by *auto* — na osnovu definicije funkcije sadrzi
finally show *?case* .
qed

7.3.1 Implementacija obilaska stabla

Redosled obilaska drveta može biti: infiksni redosled (LKD), prefiksni redosled (KLD) i postfiksni redosled (LDK), gde su velikim slovima označena Levo podstablo, Desno podstablo, i Koren stabla.

Neefikasna implementacija

Zadatak 7.49. Definisati funkciju koja od elemenata drveta kreira listu koja se dobija infiksnim obilaskom drveta.

Prilikom kreiranja liste sa desne strane se koristi nadovezivanje listi operatorom *append* tako da se element koji se nalazi u korenu drveta pretvara u listu stavljanjem u okviru uglastih zagrada. Primetimo da funkciju ne možemo nazvati *infix* jer je to rezervisana ključna reč u Isabelle/HOL. I ova funkcija se može napraviti za proizvoljni tip drveta i to bi se uradilo na isti način.

primrec *infiks* :: *nat Drvo* ⇒ *nat list* **where**
infiks Null = [] |
infiks (*Cvor l v d*) = (*infiks l*) @ [*v*] @ (*infiks d*)

value *infiks test-drvo* — testiramo funkciju infiks

Na efikasnost ove funkcije utiču funkcije koje se javljaju u njenoj definiciji. Pošto se koristi operacija *append* koja je linearna po dužini liste, za svaki element stabla očekujemo kvadratnu složenost ovog algoritma. Postavlja se pitanje da li je moguće napraviti efikasniju implementaciju - što ćemo i uraditi kasnije. Prvo ćemo proveriti neka svojstva ovako definisane funkcije funkcije.

Dokaz korektnosti

Da bismo dokazali da je implementacija obilaska stabla koju smo napisali korektna potrebno je dokazati određena svojstva koja važe za tu funkciju.

Dokazati da je skup svih elemenata liste koja se dobija infiksnim obilaskom jednak skupu svih elemenata drveta:

lemma *set-infiks-skup* [*simp*]:
set (*infiks drvo*) = *skup drvo*
by (*induction drvo*) *auto*

Primetimo da ovakvo slično tvrđenje važi i ako se koristi *multiskup*. Da bi se koristili multiskupovi potrebno je uključiti biblioteku *Multiset*. Na taj način pored funkcije set dobijamo i funkciju mset:

term *mset*
value *mset* [*1*::*nat*, *3*, *1*, *2*, *1*]

Napisati funkciju koja određuje multiskup elemenata drveta. I ova funkcija se može zapisati za proizvoljan tip drveta. Ugrađena konstanta koja označava prazan multiskup se zapisuje {#}. Simbol unije za multiskup je +. Kreiranje jednočlanog multiskupa od korenog elementa v se radi naredbom `{#v#}`.

primrec *multiskup* :: *nat Drvo* ⇒ *nat multiset* **where**
multiskup Null = {#} |
multiskup (*Cvor l v d*) = *multiskup l* + {#*v*#} + *multiskup d*

Dokazati da je multiskup liste dobijene infiksnim obilaskom jednak multiskupu svih elemenata drveta:

lemma *mset-infiks-multiskup* [*simp*]:
mset (*infiks drvo*) = *multiskup drvo*
by (*induction drvo*) *auto*

Efikasna implementacija obilaska stabla

Pokušaćemo da implementiramo optimizovanu verziju funkcije za infiksni obilazak stabla. Želimo da izbegnemo operaciju *append* koja u sebi koristi dodavanje elementa na kraj liste.

Da bi implementacija bila najefikasnija moguća potrebno je da koristimo samo operaciju dodavanja elementa na početak liste. Operacija dodavanja na početak je efikasna operacija zbog toga što su liste interno u funkcionalnim programskim jezicima implementirane kao jednostruko povezane liste i dodavanje na početak se radi u konstantnom vremenu, za razliku od dodavanja elementa na kraj liste koje se izvršava u linearnom vremenu (i zavisi od dužine liste).

Da bismo mogli ovu funkciju da implementiramo na ovako opisan način (korišćenjem samo funkcije dodavanja elementa na početak liste) trebaće nam dodatni parametar u kome ćemo malo po malo akumulirati rezultat. Slično kao što smo radili kada smo pisali funkciju za efikasno obrtanje elemenata liste.

Funkcija *infiks-opt* će imati isti potpis kao funkcija *infiks* i služice kao omotač glavnoj rekurzivnoj funkciji *infiks-opt′* i pozivaće je sa prosleđenom

praznom listom kao drugi argument. Naša glavna funkcija neće biti primitivno rekurzivna. Pomoćna funkcija će biti primitivno rekurzivna (i njena struktura će odgovarati strukturi drveta) i imaće kao drugi argument listu u kojoj se akumulira rezultat.

Ovde moramo da vodimo računa o redosledu kojim se primenjuju operacije. Želimo infiksni obilazak i jedino imamo na raspolaganju operaciju nadovezivanja na početak liste i rekurzivni poziv. Pošto je u pitanju infiksan obilazak stabla, koje odgovara (L)evom podstablu (K)orenu (D)esnom podstablu i sve to treba dopisati na akumuliranu listu xs, čitamo rekurzivni poziv zdesna na levo: prvo treba da primenimo rekurzivni poziv na desno podstablo i listu xs, nakon toga dopišemo na početak rezultujuće liste koreni element v i tako dobijenu listu prosledimo kao drugi argument drugom rekurzivnom pozivu u kojem će dopisati levo podstablo originalnog drveta na nju.

primrec *infiks-opt′* :: *nat Drvo* ⇒ *nat list* ⇒ *nat list* **where**
infiks-opt′ Null xs = *xs* |
infiks-opt′ (*Cvor l v d*) *xs* = *infiks-opt′ l* (*v* # *infiks-opt′ d xs*)

definition *infiks-opt* :: *nat Drvo* ⇒ *nat list* **where**
infiks-opt drvo = *infiks-opt′ drvo* []

value *infiks-opt test-drvo* — testiramo optimizovanu verziju

Dokaz korektnosti optimizovane verzije

Princip koji je ovde opisan se često primenjuje kod verifikacije koda. Uvek je lakše definisati jednostavniju neefikasnu verziju, za koju je lakše dokazati potrebna svojstva. Nakon toga, kada definišemo efikasniju implementaciju, dovoljno je pokazati ekvivalentnost sa osnovnom neefikasnom implementacijom i onda znamo da važe sva svojstva koja smo dokazali za neefikasnu implementaciju.

Zadatak 7.50. Dokazati da su osnovna i optimizovana verzija ekvivalentne.

Dokaz kreće od 1.0. Pogledajte ispod.

Automatski dokaz, indukcijom i auto ne prolazi. Razlog za to je što definicije date primitivnom rekurzijom ulaze u proces simplifikacije ali definicije date ključnom rečju *definition* ne ulaze u proces simplifikacije, pa takve definicije moraju ručno da se raspišu ili makar dodaju simplifikatoru. Naredba *unfolding infiks-opt-def* će raspisati definiciju funkcije *infiks-opt* i pojaviće se funkcija *infiks-opt′*.

Kada nakon toga ponovo poku[amo da primenimo auto, vidimo da smo stigli do nekih osobina samo funkcije *infiks-opt′* pa moramo da formulišemo pomoćnu lemu 1.1.

Primetimo da je formulacija pomoćne leme uopštena, iako se u narednom dokazu koristi funkcija *infiks-opt′* samo sa praznom listom kao drugi argument, formulisaćemo je opštije sa proizvoljnim argumentom xs. Vidimo da primena indukcije i automatskog dokazivača staje u trenutku kada dokazivač menja drugi argument funkcije *infiks-opt′*. Slično kao u primeru sa obrtanjem elemenata liste, da bi bila dozvoljena promena argumenta moramo upotrebiti ključnu reč *arbitrary* i pokušavamo ponovo.

Primetimo da je u ovom dokazu svejedno da li ćemo navesti i za promenljivu ys da bude proizvoljna jer se njena vrednost ne menja u toku izvršavanja dokaza.

— 1.1
lemma *infiks-opt′-append*:
shows *infiks-opt′ drvo xs* @ *ys* = *infiks-opt′ drvo* (*xs* @ *ys*)
— apply (induction drvo)
— apply auto
apply (*induction drvo arbitrary*: *xs*)
apply *auto*
done

— Krećemo odavde 1.0!
— Prilikom testiranja koda skinite komentare nad svim komandama koje su korišćene
lemma *infiks-infiks-opt*[*code*]:
— [code] označava da se prilikom generisanja koda koristi efikasnija verzija
shows *infiks drvo* = *infiks-opt drvo*
apply (*induction drvo*)
— apply auto
— ne zna da raspiše definiciju *infiks-opt*, pa je ručno raspetljavamo
unfolding *infiks-opt-def*
— apply auto
— automatski dokazivač je stao kod izraza oblika
— *infiks-opt′ drvo* [] @ - = *infiks-opt′ drvo* - pa formulišemo dodatnu lemu 1.1. koja kombinuje funkciju *infiks-opt′* i funkciju append i dodajemo je automatskom dokazivaču
apply (*auto simp add*: *infiks-opt′-append*)
done

Ova dva dokaza se mogu kraće zapisati na naredni način:

```
lemma infiks_opt'_append:
  shows "infiks_opt' drvo xs @ ys = infiks_opt' drvo (xs @ ys)"
  by (induction drvo arbitrary: xs ys) auto

lemma infiks_infiks_opt[code]:
  shows "infiks drvo = infiks_opt drvo"
  unfolding infiks_opt_def
  by (induction drvo) (auto simp add: infiks_opt'_append)
```

Dokaze ovih dveju lema možemo raspisati i u Isar-u da vidimo šta se tačno dešava u pozadini.

Kada dođemo do trenutka da želimo da primenimo induktivnu hipotezu, treba da izvučemo nekako ili prvi ili drugi deo induktivne pretpostavke. Za drugi deo bi nam bilo potrebno da nekako izvučemo deo koji se tiče desnog poddrveta (označenog sa d), odnosno *infiks-opt′ d xs* iz zagrade; a za prvi deo nam treba *infiks-opt′ l xs*, ali levo poddrvo se kombinuje sa drugom listom.

Možemo da zaključimo da je ovako formulisana induktivna hipoteza preslaba i da ne možemo da primenimo ni jedan korak dalje pa moramo da ojačamo induktivnu hipotezu zato što je ona trenutno tačna za fiksirane xs i ys (u okviru jednog koraka). Preciznije, ono što je nama ovde dovoljno jeste da ova induktivna hipoteza važi za proizvoljno xs i ys i koristi se ugrađeni mehanizam Isabelle/HOL *arbitrary*.

Dodavanjem ključne reči *arbitrary* u dokazu teoreme, kada uđemo u korak koji se dokazuje indukcijom, možemo primetiti da su xs i ys proizvoljni (što Isabelle/HOL označava znakom pitanja ispred naziva promenljive - naspram oznaka za xs i ys u prethodnom dokazu). Sada se moze primeniti prvi deo induktivne hipoteze.

lemma *infiks-opt′-append-isar*:
shows *infiks-opt′ drvo xs @ ys = infiks-opt′ drvo (xs @ ys)*
— proof (induction drvo)
— prvo pokrenite ovaj dokaz sa običnom indukcijom, pa onda sa narednom naredbom:
proof (*induction drvo arbitrary*: *xs ys*)
case *Null*
then show *?case* **by** *simp*
next
case (*Cvor l x d*)
have *infiks-opt′* (*Cvor l x d*) *xs @ ys = infiks-opt′ l* (*x # infiks-opt′ d xs*) *@ ys*
by *simp* — na osnovu definicije *infiks-opt′*
also have *... = infiks-opt′ l* (*x # infiks-opt′ d xs @ ys*)
using *Cvor*(*1*)

by *simp* — na osnovu prvog dela induktivne hipoteze, za levo podstablo
also have ... = *infiks-opt′ l* (*x* # *infiks-opt′ d* (*xs* @ *ys*))
using *Cvor*(*2*)
by *simp* — na osnovu drugo dela induktivne hipoteze, za desno podstablo
also have ... = *infiks-opt′* (*Cvor l x d*) (*xs* @ *ys*)
by *simp* — na osnovu definicije *infiks-opt′*
finally show *?case*
.
qed

U drugom dokazu opet raspisujemo definiciju *infiks-opt*.

lemma *infiks-infiks-opt-isar*:
shows *infiks drvo* = *infiks-opt drvo*
unfolding *infiks-opt-def*
proof (*induction drvo*)
case *Null*
then show *?case* **by** *simp*
next
case (*Cvor l x d*)
have *infiks* (*Cvor l x d*) = *infiks l* @ [*x*] @ *infiks d*
by *simp* — na osnovu definicije funkcije infiks koja je deo simplifikatora
also have ... = *infiks-opt′ l* [] @ [*x*] @ *infiks-opt′ d* []
using *Cvor*
by *simp* — na osnovu oba dela induktivne hipoteze
also have ... = *infiks-opt′ l* [] @ (*x* # *infiks-opt′ d* [])
by *simp* — dodavanje jednočlane liste na početak druge liste se može uraditi tarabicom
also have ... = *infiks-opt′ l* ([] @ *x* # *infiks-opt′ d* [])
by (*simp add*: *infiks-opt′-append*) — na osnovu teoreme
also have ... = *infiks-opt′ l* (*x* # *infiks-opt′ d* [])
by *simp* — dodavanje prazne liste
also have ... = *infiks-opt′* (*Cvor l x d*) []
by *simp* — na osnovu definicije *infiks-opt′*
finally show *?case*
.
qed

Provere radi, eksportujemo kod funkcije infiks u Haskell sledećom naredbom. Rezultat se moze videti u fajlu *Cas11-vezbe.hs* koji se dobija kada se u Output prozoru klikne na "See theory exports", i sa leve strane se izlaz može naći u folderu "export1".

export-code *infiks-opt* **in** *Haskell*

Nakon što smo dokazali sve ove leme, mozemo smatrati da imamo verifiko-

van kod. Ovim eksportovanjem dobijamo definiciju drveta u programskom jeziku Haskel i implementaciju optimizovane funkcije za infiksan ispis. Ono što znamo sigurno jeste da je ova funkcija korektna tj. da zadovoljava svojstva koja smo do sada dokazali.

Ovo se može uraditi i za druge programske jezike (SML, Scala), i dobijamo verifikovan kod bez bagova koji bezbedno moze da se integriše u neki drugi projekat.

export-code *infiks-opt* **in** *Scala*

7.3.2 Pretrazivačko (sortirano) drvo

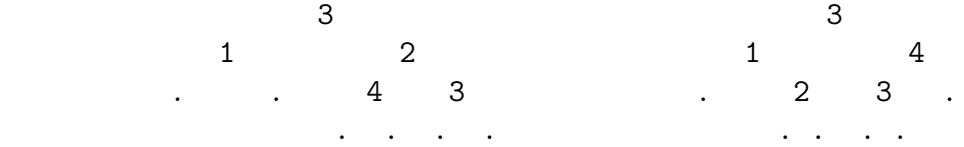
Precizna karakterizacija pretaživačkog (sortiranog) drveta: drvo je sortirano ako i samo ako važi da je i levo i desno poddrvo sortirano i da je koren drveta veći i jednak od svih elemenata iz levog poddrveta i manji ili jednak od svih elemenata desnog poddrveta.

Zadatak 7.51. Napisati funkciju koja proverava da li je dato drvo sortirano.

primrec *sortirano* **where**
sortirano Null $\longleftrightarrow$ *True* |
sortirano (*Cvor l v d*) $\longleftrightarrow$ *sortirano l* $\wedge$ *sortirano d* $\wedge$
($\forall$ $x \in$ *skup l*. $x \leq v$) $\wedge$ ($\forall$ $x \in$ *skup d*. $v \leq x$)

Testiraćemo funkciju *sortirano* na starom primeru koji smo koristili od ranije i na drvetu koje ima iste elemente ali raspoređene tako da oblikuju sortirano stablo.

Primer nesortiranog i sortiranog drveta.



value *test-drvo*
value *sortirano test-drvo*
value *infiks test-drvo*

definition *test-drvo-sortirano* :: *nat Drvo* **where**
test-drvo-sortirano = *Cvor* (*Cvor Null 1* (*Cvor Null 2 Null*)) *3* (*Cvor* (*Cvor Null 3 Null*) *4 Null*)

value *sortirano test-drvo-sortirano*
value *infiks test-drvo-sortirano*

Možemo primetiti da funkcija *sortirano* dobro radi na ova dva primera ali i da je lista koja se dobija funkcijom *infiks* sortirana, odnosno nesortirana - u zavisnosti od toga kakvo je bilo polazno drvo. Ovo ćemo i dokazati.

Zadatak 7.52. Dokazati da se infiksnim obilaskom pretraživačkog drveta dobija sortirana lista.

U dokazu ove leme ćemo koristiti *sledgehammer* da nam pomogne da nađemo pomoćne leme koje se koriste u dokazu. Te leme se tiču listi, a o njima već znamo jako puno pa nema potrebe da mi sami dokazujemo nove leme u ovom slučaju.

Već je ranije rečeno, ali podsetimo se da se *sledgehammer* ne može upotrebiti za pronalaženje dokaza koji koriste indukciju, tako da kao prvi korak mi sami primenimo indukciju. Nakon toga ponovo treba biti obazriv jer ako pozovemo odmah *sledgehammer* on će biti primenjen samo na prvi cilj, odnosno na bazu indukcije što najčešće ne želimo. Tako da ćemo i drugi korak mi sami napisati i pokušati sa automatskim dokazivačem koji uspešno rešava prvi cilj. Nakon toga nam ostaje induktivni korak koji je malo komplikovaniji.

lemma *sortirano-sorted-infiks*:
assumes *sortirano drvo*
shows *sorted* (*infiks drvo*)
using *assms*
apply (*induction drvo*)
apply *auto*
— rešava prvi podcilj, ali ne i drugi pa primenjujemo sledgehammer
— sledgehammer
using *sorted-append* **by** *fastforce*

Ovaj dokaz se uspešno završio sa predlogom koji nam je dao *sledgehammer*, ali je korišćen *fasforce*. Vežbe radi, pogledajmo šta se dešava u pozadini i kakav bismo dokaz dobili ako bismo koristili *auto* umesto toga.

Ponavljamo dokaz iste leme, s tim što smo lemu koju je *sledgehammer* našao prosleđujemo uz *auto*. Nakon toga ćemo opet pokušati sa *sledgehammer*-om i dobiti još jednu lemu koja je korišćena u dokazu.

lemma *sortirano-sorted-infiks′*:
assumes *sortirano drvo*
shows *sorted* (*infiks drvo*)
using *assms*

apply (*induction drvo*)
apply (*auto simp add*: *sorted-append*)
— sledgehammer
using *le-trans* **by** *blast*

Napomena, do ovog zaključka smo mogli i sami da dođemo ako bismo pogledali šta se dešava sa dokazivačem nakom primene automatskog dokazivača *auto*. Sistem se zaustavio na koraku oblika `sorted (_ @ _)` pa možemo iskoristiti naredbu *find-theorems* da pronađemo odgovarajuću teoremu, *sorted-append*: *sorted* (*xs* @ *ys*) = (*sorted xs* ∧ *sorted ys* ∧ (∀ *x*∈*set xs*. ∀ *y*∈*set ys*. *x* ≤ *y*)). Nakon toga ako ponovo pogledamo gde se dokazivač zaustavio, vidimo da je u pitanju tranzitivnost operacije poređenja ≤ što je sadržano u lemi *le-trans*: ⟦*i* ≤ *j*; *j* ≤ *k*⟧ ⟹ *i* ≤ *k*.

find-theorems *sorted* (- @ -)
thm *sorted-append*

thm *le-trans*

lemma *sortirano-sorted-infiks''*:
assumes *sortirano drvo*
shows *sorted* (*infiks drvo*)
using *assms*
apply (*induction drvo*)
apply (*auto simp add*: *sorted-append*)
apply (*auto simp add*: *le-trans*)
done

ili skraćeno:

lemma *sortirano-sorted-infiks'''*:
assumes *sortirano drvo*
shows *sorted* (*infiks drvo*)
using *assms*
by (*induction drvo*) (*auto simp add*: *sorted-append le-trans*)

Zadatak 7.53. Napisati funkciju koja ubacuje element u pretraživačko drvo.

primrec *ubaci* :: *nat* ⇒ *nat Drvo* ⇒ *nat Drvo* **where**
ubaci v Null = (*Cvor Null v Null*) |
ubaci v (*Cvor l v' d*) =
(*if v* ≤ *v' then*
Cvor (*ubaci v l*) *v' d*

else
Cvor l v′ (ubaci v d))

Zadatak 7.54. Dokazati da nakon ubacivanja elementa x drvo sadrži element x.

Napomena: U ovim dokazima, alternativa tome da leme klasifikujemo kao simplifikatorske ključnom rečju [*simp*] koju pišemo odmah nakon naziva leme, jeste da ih eksplicitno navodimo kao što smo radili na prethodnim časovima.

lemma *sadrzi-ubaci* [*simp*]:
shows *sadrzi (ubaci x drvo) x*
by (*induction drvo*) *auto*

Zadatak 7.55. Dokazati da se skup, multiskup i zbir elemenata drveta uvećavaju za element x nakon ubacivanja x u drvo.

lemma *skup-ubaci* [*simp*]:
shows *skup (ubaci x drvo) = {x} ∪ skup drvo*
by (*induction drvo*) *auto*

lemma *multiskup-ubaci*[*simp*]:
shows *multiskup (ubaci x drvo) = {#x#} + (multiskup drvo)*
by (*induction drvo*) *auto*

lemma *zbir-ubaci* [*simp*]:
shows *zbir (ubaci x drvo) = x + zbir drvo*
by (*induction drvo*) *auto*

Zadatak 7.56. Dokazati da funkcija *ubaci* očuvava sortiranost drveta.

lemma *sortirano-ubaci* [*simp*]:
assumes *sortirano drvo*
shows *sortirano (ubaci x drvo)*
using *assms*
by (*induction drvo*) *auto*

Nakon što smo dokazali ovu lemu možemo iskoristiti funkciju *ubaci* za kreiranje sortiranog drveta na osnovu elemenata liste:

primrec *listaUDrvo* :: *nat list* ⇒ *nat Drvo* **where**
listaUDrvo [] = *Null* |
listaUDrvo (x # xs) = ubaci x (listaUDrvo xs)

value *listaUDrvo* [*3*, *4*, *7*, *2*, *1*, *6*] — testiramo funkciju listaUDrvo

Dokazati da se skup elemenata nije promenio, tj. da je skup elemenata drveta dobijenog od liste jednak skupu elemenata liste. Isto to pokazati i za multiskup i za zbir.

lemma [*simp*]: *skup* (*listaUDrvo xs*) = *set xs*
by (*induction xs*) *auto*

lemma [*simp*]: *multiskup* (*listaUDrvo xs*) = *mset xs*
by (*induction xs*) *auto*

lemma [*simp*]: *zbir* (*listaUDrvo xs*) = *sum-list xs*
by (*induction xs*) *auto*

Dokazati da je ovako kreirano drvo pretraživačko drvo.

lemma [*simp*]: *sortirano* (*listaUDrvo xs*)
by (*induction xs*) *auto*

Sve do sada kreirane funkcije i dokazane leme nam omogućavaju da opišemo *TreeSort* postupak sortiranja lista tako što od elemenata liste formiramo drvo, pa ga infiksno obiđemo.

definition *sortiraj* :: *nat list* ⇒ *nat list* **where**
sortiraj xs = *infiks* (*listaUDrvo xs*)

Testiramo funkciju sortiraj

value *sortiraj* [*3*, *4*, *7*, *2*, *1*, *6*]

Eksportujemo kod u Haskell

export-code *sortiraj* **in** *Haskell*

Dokazati da je ovako kreirana lista sortirana, tj. dokazati da funkcija sortiraj daje sortiranu listu.

theorem *sorted* (*sortiraj xs*)
unfolding *sortiraj-def*
by (*simp add*: *sortirano-sorted-infiks*)

Dokazati da funkcija sortiraj čuva skup elemenata liste i multiskup elemenata liste.

theorem *set* (*sortiraj xs*) = *set xs*
unfolding *sortiraj-def*
by *simp*

theorem *mset* (*sortiraj xs*) = *mset xs*
unfolding *sortiraj-def*
by *simp*

Napisati funkciju koja određuje maksimalni element neuređenog drveta - primer funkcije koja nije definisana primitivnom rekurzijom.

Ako funkcija koju definišemo nema strogo podeljene slučajeve prazne liste ([]) i liste koja ima makar jedan element (*x1 # xs*) onda ne možemo da koristimo primitivnu rekurziju.

Napomena: U ovom primeru se koristi konstrukcija *fun* koja je mnogo naprednija konstrukcija od *primrec* i podržava generalnu rekurziju. Kod *primrec* zaustavljanje automatski sledi, a kod *fun* se mora dokazati zaustavljanje (i to se najčešće radi automatski, korišćenjem ugrađenog termination checker-a). Pogledajte izlaz u Isabelle/HOL, ispisaće: *Found termination order*. U opštem slučaju, zaustavljanje se dokazuje pokazivanjem da su argumenti rekurzivnih poziva na neki način manji od originalnih argumenata. U ovom poglavlju ćemo se ograničiti na funkcije za koje Isabelle/HOL može sam automatski da dokaže zaustavljanje.

```
fun maks :: nat Drvo ⇒ nat where
  maks (Cvor Null v Null) = v
| maks (Cvor l v Null) = max (maks l) v
| maks (Cvor Null v d) = max v (maks d)
| maks (Cvor l v d) =
    (let maksl = maks l;
         maksd = maks d
     in max (max maksl v) maksd)
```

value *maks* (*listaUDrvo* [*3*, *4*, *1*, *7*, *6*, *3*, *2*]) — testiramo funkciju maks

Korektnost funkcije maks - dokaz ostavljamo za kasnije

```
declare [[quick-and-dirty=true]]
lemma
  assumes sortirano drvo drvo ≠ Null
  shows ∀ x ∈ skup drvo. maks drvo ≥ x
  using assms
  sorry
```

7.3.3 Provera da li je lista sortirana

Definisaćemo slične funkcije za rad sa listama zato što su liste jednostavniji tip podataka i pokazaćemo nekoliko lema čiji dokazi će biti jednostavniji

nego sa stablima.

Zadatak 7.57. Definisati naše funkcije koje određuju prvi, odnosno poslednji element liste.

primrec *glava* :: *nat list* ⇒ *nat* **where**
glava (*x1* # *xs*) = *x1*

primrec *kraj* :: *nat list* ⇒ *nat* **where**
kraj (*x1* # *xs*) = (*if xs* = [] *then x1 else kraj xs*)

Zadatak 7.58. Definisati našu funkciju koja proverava da li je lista sortirana.

Lista je sortirana (rastuce) ako i samo ako je sortiran rep liste i ako je njen prvi element manji ili jednak od svih ostalih elemenata liste. Pošto imamo pretpostavku da je rep liste takođe sortiran onda je dovoljno da prvi element liste bude manji ili jednak od glave repa liste. U ovoj definiciji možemo koristiti našu funkciju:

primrec *sortirana′* :: *nat list* ⇒ *bool* **where**
sortirana′ [] ⟷ *True* |
sortirana′ (*x1* # *xs*) ⟷ *sortirana′ xs* ∧ (*if xs* = [] *then True else x1* ≤ *glava xs*)

Drugi način je da koristimo ugrađenu funkciju *hd* koja vraća prvi element liste i kvantifikator kojim ćemo opisati da je prvi element liste manji ili jednak od svih elemenata koji se nalaze u repu:

term *hd*

primrec *sortirana* :: *nat list* ⇒ *bool* **where**
sortirana [] ⟷ *True* |
sortirana (*x1* # *xs*) ⟷ *sortirana xs* ∧ (∀ *x* ∈ *set xs*. *x1* ≤ *x*)

value *sortirana* [*1*,*2*,*3*]
value *sortirana* [*2*,*1*,*3*]

Sada ćemo dokazati da je u sortiranoj listi prvi element liste manji ili jednak od svih ostalih elemenata liste.

Kako glava liste ne postoji za prazne liste, pretpostavka da je lista sa kojom radimo neprazna mora biti eksplicitno dodata prilikom formulisanja naredne leme. Ovaj dokaz automatski prolazi.

lemma *sortirana-glava-najmanja*:
assumes *xs* ≠ [] *sortirana xs*

shows $\forall\ x \in set\ xs.\ hd\ xs \leq x$
using *assms*
apply (*induction xs*)
apply *auto*
done

Pogledajmo kako bi izgledao dokaz ove leme kada bi se koristila prva definicija, gde se upoređuje prvi element liste. Iskoristićemo sledgehammer i primetimo da je dokaz znatno komplikovaniji zato što više odstupamo od kvantifikatora nego što je neophodno. Tako da nadalje nećemo koristiti prvu definiciju funkcije sortirano nego samo drugu.

lemma *sortirana-glava-najmanja-def'*:
assumes $xs \neq []$ *sortirana' xs*
shows $\forall\ x \in set\ xs.\ hd\ xs \leq x$
using *assms*
apply (*induction xs*)
apply *auto*
by (*metis glava.simps le-trans list.discI list.sel(1) list.set-cases*)

Potrebno je dokazati i da su ove dve definicije ekvivalentne (uz pomoć sledgehammer-a:

lemma *sortirana xs* $\longleftrightarrow$ *sortirana' xs*
apply (*induction xs*)
apply *auto*
apply (*metis glava.simps list.collapse list.set-sel(1)*)
by (*metis glava.simps le-trans list.sel(1) list.set-cases sortirana-glava-najmanja-def'*)

Možemo sada formulisati sledeće tvrđenje koje se odnosi samo na liste koje imaju makar jedan element:

lemma *sortirana-Cons*:
sortirana $(x \# xs) \longleftrightarrow$ *sortirana xs* $\wedge$ $(\forall\ x' \in set\ xs.\ x \leq x')$
by (*induction xs*) *auto*

Sada ako bismo želeli da dokažemo ovu lemu u isaru, granaćemo po slučajevima da li je xs prazna ili neprazna lista. Iskomentarisaćemo detaljno samo prvu granu dokaza.

lemma *sortirana-Cons-isar*:
sortirana $(x \# xs) \longleftrightarrow$ *sortirana xs* $\wedge$ $(\forall\ x' \in set\ xs.\ x \leq x')$
proof (*cases xs* $= []$)
case *True*
thus *?thesis*
by *simp*
next

```
  case False
  show ?thesis
  proof — pustićemo dokazivač da ekvivalenciju razbije na dve implikacije
    assume sortirana (x # xs)
   hence sortirana xs x ≤ hd xs — na osnovu definicije sortirana, kako je x manji
ili jednak od svakog elementa liste xs, biće manji ili jednak od prvog elementa u
listi
      using ⟨xs ≠ []⟩
      by simp-all
— primenjujemo simp-all zato što imamo dva cilja u ovom koraku
— da smo koristili definiciju sa funkcijom glava ovaj korak ne bi prošao, lakše je
nekada sa kvantifikatorima
    hence ∀ x′ ∈ set xs. hd xs ≤ x′
— na osnovu ranije dokazane leme, glava neprazne liste je manja ili jednaka od
svih njenih elemenata
      using sortirana-glava-najmanja[of xs] ⟨xs ≠ []⟩
— instanciranje ove leme nije neophodno, dodato je samo zbog čitljivosti
      by simp
— sada znamo da je x manji ili jednak od prvog elementa od xs, i da je prvi
element od xs manji ili jednak od svih elemenata liste xs; na osnovu tranzitivnosti
operacije manje ili jednako možemo da zaključimo da je x manji ili jednak od svih
elemenata liste xs
    hence ∀ x′ ∈ set xs. x ≤ x′
      using ⟨x ≤ hd xs⟩
      by auto
    thus sortirana xs ∧ (∀ x′ ∈ set xs. x ≤ x′)
      using ⟨sortirana xs⟩
      by simp
  next
— druga grana dokaza
    assume sortirana xs ∧ (∀ x′ ∈ set xs. x ≤ x′)
    hence sortirana xs ∧ x ≤ hd xs
      using ⟨xs ≠ []⟩
      by simp
      thus sortirana (x # xs)
       using sortirana-glava-najmanja[of xs] ⟨xs ≠ []⟩
       by auto
  qed
qed
```

Zadatak 7.59. Dokazati da se nadovezivanjem dveju sortiranih listi dobija sortirana lista ako i samo ako je svaki element prve liste manji ili jednak od svakog elementa druge liste.

lemma *sortirana-append*:
sortirana (*xs* @ *ys*) ⟷ *sortirana xs* ∧ *sortirana ys* ∧
(∀ *x* ∈ *set xs*. ∀ *y* ∈ *set ys*. *x* ≤ *y*)
apply (*induction xs*)
apply *auto*
done

I odgovarajući dokaz u isar-u:

lemma *sortirana-append-isar*:
sortirana (*xs* @ *ys*) ⟷ *sortirana xs* ∧ *sortirana ys* ∧
(∀ *x* ∈ *set xs*. ∀ *y* ∈ *set ys*. *x* ≤ *y*)
proof (*induction xs*)
case *Nil*
thus *?case*
by *simp*
next
case (*Cons x xs*)
show *?case*
proof (*cases xs* = [])
case *True*
thus *?thesis*
using *sortirana-Cons*[*of x ys*]
by *simp*
next
case *False*
show *?thesis*
proof−
have *sortirana* ((*x* # *xs*) @ *ys*) ⟷
sortirana (*x* # (*xs* @ *ys*))
by *simp*
also have ... ⟷ *sortirana* (*xs* @ *ys*) ∧ (∀ *x′* ∈ *set* (*xs* @ *ys*). *x* ≤ *x′*)
using *sortirana-Cons*[*of x xs* @ *ys*]
by *simp*
also have ... ⟷ *sortirana xs* ∧ *sortirana ys* ∧ (∀ *x′* ∈ *set xs*. ∀ *y* ∈ *set ys*.
x′ ≤ *y*) ∧ (∀ *x′* ∈ *set* (*xs* @ *ys*). *x* ≤ *x′*)
using *Cons*
by *simp*
also have ... ⟷ (*sortirana xs* ∧ (∀ *x′* ∈ *set xs*. *x* ≤ *x′*)) ∧ *sortirana ys* ∧
(∀ *x*∈*set* (*x* # *xs*). ∀ *y*∈*set ys*. *x* ≤ *y*)
by *auto*
also have ... ⟷ *sortirana* (*x* # *xs*) ∧ *sortirana ys* ∧ (∀ *x*∈*set* (*x* # *xs*).
∀ *y*∈*set ys*. *x* ≤ *y*)
using *sortirana-Cons*[*of x xs*]

```
      by simp
    finally
    show ?thesis
      .
  qed
 qed
qed
```

7.3.4 Rotiranje drveta

Pod rotiranim drvetom podrazevamo drvo sa istim elementima kao polazno drvo, s tim što su elementi u novom drvetu raspoređeni tako da odgovaraju slici u ogledalu početnom drvetu (postavljenom sa strane).

Zadatak 7.60. Definisati funkciju koja rotira drvo i dokazati da se dvostrukim rotiranjem dobija originalno drvo. Napisati i automatski i isar dokaz.

```
primrec mirror :: 'a Drvo ⇒ 'a Drvo where
mirror Null = Null |
mirror (Cvor l v d) = Cvor (mirror d) v (mirror l)

lemma mirror-id: mirror (mirror t) = t
 apply (induction t)
  apply auto
 done

lemma mirror-id-isar: mirror (mirror t) = t
proof (induction t)
 case Null
 show ?case by simp
next
 case (Cvor t1 x2 t2)
 have mirror (mirror (Cvor t1 x2 t2)) = mirror (Cvor (mirror t2) x2 (mirror
t1)) by simp
 also have ... = Cvor (mirror (mirror t1)) x2 (mirror (mirror t2)) by simp
 also have ... = Cvor t1 x2 t2 using Cvor by simp
 finally show ?case .
qed
```

Zadatak 7.61. Definisati funkciju koja pravi od drveta listu dobijenu postfiksnim obilaskom drveta i funkciju koja pravi od drveta listu dobijenu prefiksnim obilaskom drveta.

Dokazati da se prefiksnim obilaskom rotiranog stabla dobija obrnuta lista u odnosu na postfiksni obilazak originalnog stabla.

primrec *prefiks* :: *nat Drvo* ⇒ *nat list* **where**
prefiks Null = [] |
prefiks (*Cvor l v d*) = [*v*] @ *prefiks l* @ *prefiks d*

primrec *postfiks* :: *nat Drvo* ⇒ *nat list* **where**
postfiks Null = [] |
postfiks (*Cvor l v d*) = *postfiks l* @ *postfiks d* @ [*v*]

value *test-drvo*
value *prefiks test-drvo*
value *mirror test-drvo*
value *prefiks* (*mirror test-drvo*)
value *postfiks test-drvo*
value *rev* (*postfiks test-drvo*)
value *rev* (*postfiks test-drvo*) = *prefiks* (*mirror test-drvo*)

lemma *prefiks-mirror-postfiks*:
shows *prefiks* (*mirror t*) = *rev* (*postfiks t*)
apply (*induction t*)
apply *auto*
done

Raspisati i dokaz u Isar-u.

lemma *prefiks-mirror-postfiks-isar*:
shows *prefiks* (*mirror t*) = *rev* (*postfiks t*)
proof (*induction t*)
case *Null*
show *?case* **by** *simp*
next
case (*Cvor l x d*)
have *prefiks* (*mirror* (*Cvor l x d*)) = *prefiks* (*Cvor* (*mirror d*) *x* (*mirror l*))
by *simp* — na osnovu definicije mirror
also have ... = [*x*] @ *prefiks* (*mirror d*) @ *prefiks* (*mirror l*)
by *simp* — na osnovu definicije prefiks
also have ... = [*x*] @ *rev* (*postfiks d*) @ *rev* (*postfiks l*)
using *Cvor*
by *simp* — na osnovu induktivne pretpostavke, obe grane se koriste

```
  also have ... = [x] @ rev (postfiks l @ postfiks d)
    by simp — na osnovu osobina listi
  also have ... = rev [x] @ rev (postfiks l @ postfiks d)
    by simp — na osnovu osobina listi
  also have ... = rev (postfiks l @ postfiks d @ [x])
    by simp  — na osnovu osobina listi
  also have ... = rev (postfiks (Cvor l x d))
    by simp — na osnovu definicije postfiks
  finally show ?case
    .
qed
end
```

7.4 Opšta rekurzija i algoritmi sortiranja

theory *Cas12-vezbe*
imports *Main HOL−Library.Code-Target-Nat HOL−Library.Multiset*

begin

Do sada smo samo videli funkcije definisane primitivnom rekurzijom u kojima rekurzija direktno prati strukturu tipa podataka po kome se rekurzija definiše (koristi se ključna reč `primrec`):

- Kod prirodnih brojeva razlikuju se slučaj 0 i sledbenika (Suc n)
- Kod lista se razlikuje slučaj prazne liste ([]) i nadovezanog elementa na početak liste ($x \# xs$)
- Kod drveta (koje smo sami definisali) razlikuje se slučaj praznog drveta (Null) i nepraznog čvora koji se sastoji od levog postabla, korena i desnog podstabla (Cvor l k d)

Pored primitivne rekurzije postoji mogućnost korišćenja opšte (generalne) rekurzije koja se uvodi uz pomoć ključnih reči *fun* i *function*. Osnovna razlika je u tome što definicije koje koriste ključnu reč *fun* imaju mogućnost da se njihovo svojstvo zaustavljanja automatski dokaže uz pomoć Isabelle/HOL. U situacijama kada ovako nešto nije moguće potrebno je koristiti ključnu reč *function* i eksplicitno navesti dokaz korektnosti takve definicije (što će biti naknadno objašnjeno).

Zadatak 7.62. Definisati funkciju stepenovanja.

Prvo ćemo napisati verziju uz pomoć primitivne rekurzije i njeno izvršavanje je relativno neefikasno i sporo se izvršava. Razlog za to je pre svega u internoj reprezentaciji prirodnih brojeva jer se oni predstavljaju kao 0, Suc 0, Suc (Suc 0),... pa vidimo da nam već sama ta interna reprezentacija ne odgovara u situaciji kada dobijamo velike vrednosti brojeva jer će funkcija Suc biti pozvana veliki broj puta. Rešenje je da koristimo drugačiju reprezentaciju prirodnih brojeva i koristimo mašinske tipove podataka. Uključujemo teoriju *HOL−Library.Code-Target-Nat* i koristimo reprezentaciju prirodnih brojeva koja se koristi i u ciljnom jeziku. Isabelle/HOL je programiran u ML-u i korišćenjem ove reprezentacije znatno brže se izvršava ova ista funkcija. U Isabelle/HOL nemamo prekoračenje tipa, pa o tome ne moramo da vodimo računa.

Napomena: dokazi nekih lema će se malo razlikovati u odnosu na dokaze sa profesorovog sajta, na par mesta kod profesora leme su dodate u simplifikator a u ovom dokumentu smo to najčešće izbegavali. Takođe par definicija imaju promenjene nazive. Sve ovo nisu suštinske promene.

primrec *pow′* :: *nat* ⇒ *nat* ⇒ *nat* **where**
pow′ x 0 = 1 |
pow′ x (*Suc n*) = *x* ∗ *pow′ x n*

value *pow′ 2 15*

Vežbe radi, dokazaćemo da je ova funkcija dobro definisana, odnosno da stvarno računa stepen broja. Dokaz pokušavamo standardno indukcijom i automatskim dokazivačem što prolazi.

lemma *pow′-stepen*:
pow′ x n = x ^ n
by (*induction n*) *auto*

U realnom programiranju javljaju se i mnogo bogatiji oblici rekurzije. Ilustujmo ih kroz nekoliko primera. Za početak ćemo napisati efikasniju verziju iste funkcije.

7.4.1 Efikasno stepenovanje

Razmotrimo narednu definiciju efikasnog stepenovanja. U njoj koristimo definiciju opšte rekurzije i koristimo ključnu reč *fun*. Koristićemo matematičko tvrđenje po kome, za paran broj n važi $x^n = (x^2)^{ndiv2}$. Otuda, u rekurzivnom pozivu `pow (x*x) (n div 2)` svođenje nije sa sledbenika na prethodnika pa stoga nije mogla biti upotrebljena primitivna rekurzija. Umesto toga upotrebićemo opštu rekurziju i ključnu reč *fun*.

fun *pow″* :: *nat* ⇒ *nat* ⇒ *nat* **where**
pow″ x 0 = 1 |
pow″ x n = (*if n mod 2 = 0 then pow″* (*x∗x*) (*n div 2*) *else x ∗ pow″ x* (*n−1*))

Ovaj oblik podseća izgledom na definiciju primitivne rekurzije, ali nije potrebno koristiti tako nešto već možemo preći skroz na if-then-else konstrukciju i dobiti narednu funkciju.

fun *pow* :: *nat* ⇒ *nat* ⇒ *nat* **where**
pow x n =
(*if n = 0 then 1*
else if n mod 2 = 0 then
pow (*x ∗ x*) (*n div 2*)

```
  else
    x * pow x (n - 1))
```

value *pow 2 10*

thm *pow.induct*

Nakon definisanja funkcije u Isabelle/HOL, u pozadini se automatski dokazuje teorema indukcije koja odgovara toj funkciji i koja se može koristiti u dokazu korektnosti te funkcije. Teorema se dobija kada se na ime funkcije nadoveže `.induct` i u ovom konretnom slučaju ona se zove *pow.induct*, i izgleda ovako:

$$\begin{array}{l}(\bigwedge x\ n.\ [\![[\![n \neq 0;\ n\ mod\ 2 = 0]\!] \Longrightarrow P\ (x * x)\ (n\ div\ 2);\\ \qquad [\![n \neq 0;\ n\ mod\ 2 \neq 0]\!] \Longrightarrow P\ x\ (n - 1)]\!]\\ \qquad \Longrightarrow P\ x\ n) \Longrightarrow\\ P\ a0\ a1\end{array}$$

Ova teorema suštinski tvrdi da ako pretpostavimo da svaki rekurzivni poziv (imamo ih dva) računa stepen korektno, da će onda i tekući rekurzivni poziv korektno izračunati stepen funkcije. Ova teorema se mora eksplicitno navesti u dokazu teoreme ako želimo da je koristimo, kao što ćemo videti malo kasnije.

Kada se radi sa opšte rekurzivnim funkcijama, mora se voditi računa o još jednom pravilu koje se automatski generiše i koje se automatski prosleđuje simplifikatoru. Ime tog pravila se dobija kada se na ime rekurzivne funkcije dopiše `.simps` i glasi *thm pow.simps*. Može se desiti da postajanje ovakvih pravila dovede simplifikator u beskonačnu petlju. Ako se tako nešto desi potrebno je izbaciti to pravilo iz procesa simplifikacije narednom naredbom:

```
declare pow.simps [simp del]
```

Prilikom ovakvog menjanja znanja koje poseduje simplifikator treba voditi računa da ovo može uticati na neke korake koji su ranije bili trivijalni zato što je simplifikator posedovao više znanja. Ovo pravilo, u ovom slučaju, izgleda ovako:

```
pow x n =
(if n = 0 then 1
 else if n mod 2 = 0 then pow (x * x) (n div 2) else x * pow x (n - 1))
```

thm *pow.simps*
declare *pow.simps* [*simp del*]

Dokazi svojstava primitivno-rekurzivnih podataka se dokazuju strukturnom indukcijom po odgovarajućem tipu podataka. Na sličan način se dokazi generalno-rekurzivnih funkcija izvode specijalnim oblikom indukcije koji je prilagođen toj rekurzivnoj funkciji. Korektnost tako definisanog tvrđenja se svodi na to da se iz pretpostavki da svaki rekurzivni poziv radi korektno dokaže da glavni poziv radi korektno. Ilustrujmo ovo na primeru dokazivanja korektnosti efikasne funkcije stepenovanja.

Napomena: prvo se dokazuje naredno tvrđenje, pa ako sami pokušavate da dokažete, krenite od naredne leme.

U ovom pasusu je izdvojen dokaz matematičkog tvrđenja koje stoji iza definicije koju smo već dali. Ovo je centralni argument korektnosti definicije i zasniva se na narednoj lemi koju dokazujemo zasebno.

Napomenimo da je ovde neophodno navesti tip za promenljive koje se javljaju u tvrđenju. Ovo možemo zaključiti korišćenjem naredbe

```
using [[show_types]]
```

koju pokrećemo pre dokaza teoreme koja bi ispisala (pre nego što dodamo red koji počinje sa ključnom rečju fixes) da ne može da odredi tip promenljive x. Sada možemo pozvati sledgehammer i dobiti izdvojen dokaz ovog matematičkog tvrđenja.

lemma *pow-div2*:
  fixes *x n* :: *nat*
  assumes *n mod 2 = 0*
  shows *(x * x) ^ (n div 2) = x ^ n*
— *using* [[*show-types*]]
  using *assms*
  by (*metis dvd-div-mult-self minus-mod-eq-div-mult minus-mod-eq-mult-div mod-0-imp-dvd power-mult-distrib semiring-normalization-rules*(*36*))

Dokazati da naša generalno-rekurzivna funkcija pow zaista izračunava stepen prirodnog broja. Ako bismo sada primenili običnu indukciju dobili bismo kao rezultat dva slučaja koja standardno odgovaraju nuli, i sledbeniku prirodnog broja. Ali ovakav oblik indukcije ne odgovara definiciji tipa i želimo da primenimo drugačiji tip indukcije što nam Isabelle/HOL i omogućava korišćenjem samog tipa. Koristimo naredbu:

```
proof (induction x n rule: pow.induct)
```

Pošto funkcija ima dva argumenta, moramo sve argumente te funkcije navesti (onim redosledom kojim se javljaju u definiciji), nakon čega sledi ime teoreme indukcije koja odgovara ovoj funkciji. Pošto je cela struktura

ove funkcije malo komplikovanija, ovaj dokaz ne prolazi automatski pa ćemo odmah raspisati dokaz u Isar-u.

Prvi deo dokaza prolazi relativno jednostavno dok ne dođemo do slučaja kada je n paran broj (pogledajte dokaz do tog koraka). Pogledajmo šta se dešava kada sada primenimo simplifikator (ali postupamo kao u automatskom dokazivanju i koristimo ključnu reč *apply* umesto *by*). Izdvojićemo nekoliko zaključaka do kojih možemo doći (nastavite kroz dokaz korak po korak i najbolje da pokušate sami da ga napišete prateći uputstva iz teksta):

(1) Primećujemo da simplifikator ne vidi činjenicu da n nije jednako 0 pa tu činjenicu eksplicitno dodajemo naredbom *using*.

(2) Na ovom mestu možemo da primetimo da (pre brisanja pow.simps iz simplifikatora), možemo da dobijemo beskonačnu petlju rekurzivnih poziva. Nakon brisanja pow.simps simplifikator ne može ništa da uradi pa inicijalizujemo tvrđenje za konkretne promenljive koje se trenutno javljaju u dokazu.

(3) Nakon toga primećujemo da smo sveli slučaj na rekurzivni poziv i pozivamo se na induktivnu pretpostavku i to na njen prvi deo. Ovde se induktivna hipoteza zove 1 i njen prvi deo se dobija sa 1(1).

(4) Sada smo došli do čisto matematičkog tvrđenja: kada $(x^2)^{nmod2} = x^n$, pa možemo ili upotrebiti sledgehammer

(5) ili formulisati našu novu lemu u kojoj ćemo dokaz tog matematičkog tvrđenja izvaditi iz dokaza ove glavne leme - pogledajte dokaz prethodne teoreme.

```
lemma pow-stepen:
  pow x n = x ^ n
proof (induction x n rule: pow.induct)
— ovo je način da primenimo rekurzivno pravilo specifično za funkciju pow
  case (1 x n) — ovako je označena induktivna hipoteza
  show ?case
— glavni deo dokaza analizira jedan po jedan slučaj koji se javlja u definiciji funkcije i granamo isto kao što smo granali u definiciji
  proof (cases n = 0)
    case True
— slučaj kada je n = 0 je trivijalan dok je pow.simps tvrđenje uključeno u simplifikator, kada ga izbacimo ovaj korak više ne prolazi samo sa simp pa tvrđenje koje se koristi moramo eksplicitno dodati
    thus ?thesis
      by (simp add: pow.simps)
  next
```

```
    case False
— ako n nije 0 onda opet granamo prema tome da li je n parno ili neparno
    show ?thesis
    proof (cases n mod 2 = 0)
      case True
      then show ?thesis
— pre brisanja:
— ako bismo na ovom mestu pokušali samo sa simp (pre brisanja pow.simps),
videli bismo da je simplifikator upao u beskonačnu petlju rekurzivnih poziva
— nakon brisanja:
— nakon što obrišemo pow.simps simplifikator ne može ništa da uradi pa nam
treba rešenje između ova dva ekstrema, i to rešenje dobijamo kada inicijalizujemo
tvrđenje za konkretne promenljive koje se trenutno javljaju u dokazu.
        using ‹n ≠ 0›
        using pow.simps[of x n]
        using 1(1)
— apply simp
— sledgehammer
— by (metis div-mult-self1-is-m mod-eq-0D power2-eq-square power-mult zero-less-numeral)
        by (simp add: pow-div2)
    next
      case False
— ako je n neparno pokusavamo slično, sada ćemo koristiti drugi deo induktivne
hipoteze i opet pokušavamo sa sledgehammerom koji nalazi teoremu power-eq-if
koja je već dokazana u Isabelle/HOL pa nema potrebe da je dodajemo i sami
dokazujemo
      then show ?thesis
        using ‹n ≠ 0›
        using pow.simps[of x n]
        using 1(2)
        — sledgehammer
        by (simp add: power-eq-if)
    qed
  qed
qed
```

U ovakvim dokazima može se desiti da automatski dokaz ne možemo da formulišemo, odnosno ako pokušamo da kreiramo kraći automatski dokaz sa informacijama do kojih smo došli dobili bismo dokaz narednog oblika koji iz nekog razloga ulazi u beskonačnu petlju. Na ovaj način možemo da vidimo neophodnost interaktivnih dokaza u situacijama kada automatski dokazivači ne znaju da se snađu sa informacijama koje su im neophodne za kreiranje dokaza.

```
lemma pow_stepen_auto:
  "pow x n = x ^ n"
  using pow.simps[of x n]
  apply (induction x n rule: pow.induct, auto simp: pow_div2 power_eq_if)
```

7.4.2 NZD

U ovom poglavlju ćemo videti kako možemo da isprogramiramo Euklidov algoritam za nalaženje najvećeg zajedničkog delioca dva broja.

Zadatak 7.63. Definisati osnovnu varijantu Euklidovog algoritma preko oduzimanja.

Ponovo koristimo opštu rekurziju. Kada koristimo ključnu reč *fun* sistem pokušava automatski da dokaže zaustavljanje za sve moguće ulaze. Kod primitivne rekurzije ovako nešto nije neophodno (jedini rekurzivni poziv u tom slučaju je bilo svođenje na prethodnika, što znači da smo automatski imali smanjivanje argumenta funkcije dok ne stignemo do nule što je izlaz iz rekurzije). Kada radimo sa opštom rekurzijom moramo biti veoma obazrivi zato što je moguće napraviti beskonačne cikluse. Neophodno je da se automatski može dokazati zaustavljanje ove funkcije.

Pogledajmo prvo kako izgleda nepotpuna definicija ove funkcije (dodaćemo apostrofe nakon naziva funkcije da bismo je razlikovali od konačne funkcije). Ako koristimo ključnu reč *fun* Isabelle/HOL će ispisati grešku "Could not find lexicographic termination order" što znači da nije uspeo da dokaže zaustavljanje. Sada možemo pokušati sami da ga ubedimo da se naša funkcija zaustavlja, ali za to je potrebno da koristimo ključnu reč *function* i naredbu `by pat_completeness auto`. Nakon toga potrebno je identifikovati jednu vrednost koja će se smanjivati kroz rekurzivne pozive ove naše definicije. Ako pogledamo definiciju, vrednost koja se smanjuje u svakom pozivu neće biti posebno ni prvi ni drugi argument (zato što se oni naizmenično smanjuju), ali njihov zbir se konstantno smanjuje (bez obzira na to koji argument se smanjio) i njega ćemo koristiti.

Nakon toga ključnom rečju *termination* ulazimo u proces dokazivanja da se ova funkcija zaustavlja (možemo da primetimo da Isabelle/HOL formira cilj koji treba da se dokaže) i koristimo isti princip koji smo koristili ranije i pokušavamo uz pomoć automatskih dokazivača. Korišćenjem naredbe `apply (relation "measure (\<lambda> (a, b). a + b)")` koristimo automatski dokazivač *auto*, metod *relation*, ključnu reč *measure* i nakon nje anonimnu funkciju koja od para argumenata dobijenih ovom funkcijom kreira vrednost za koju tvrdimo da ona konstantno opada.

Nakon toga pokušavamo sa automatskim dokazivačem, samo *auto* i primećujemo da Isabelle/HOL traži da pokažemo da su i a i b pozitivni. Ako pogledamo samu definiciju funkcije vidimo da nula kao argument može da dovede do beskonačne petlje (po definiciji nzd"(5,0) = nzd"(5,0) zato što je 5 veće od 0). To znači da trenutna defincija nije dobra i da ne pokriva adekvatno slučaj kada je b = 0. Slično za nzd"(0,5) = nzd"(0,5) daje beskonačnu petlju.

Ovo je primer kako automatski dokazivač može da nam pomogne i pronađe greške koje postoje u našem kodu.

Napomena: naredni kod odkomentarišite da biste ga istestirali.

```
(* fun nzd'' :: "nat \<Rightarrow> nat \<Rightarrow> nat" where *)
function nzd'' :: "nat \<Rightarrow> nat \<Rightarrow> nat" where
  "nzd'' a b =
     (if a > b then
        nzd'' (a - b) b
      else if b > a then
        nzd'' a (b - a)
      else
        a)"
  by pat_completeness auto
termination
  apply (relation "measure (\<lambda> (a, b). a + b)")
  apply auto

(* value "nzd'' 8 4" *)
```

Pa sada proširujemo definiciju funkcije za slučajeve kada je jedan od argumenata jednak nuli i sada auto prolazi.

function $nzd'' :: nat \Rightarrow nat \Rightarrow nat$ **where**
$nzd''\ a\ b =$
$(if\ a = 0\ then\ b$
$else\ if\ b = 0\ then\ a$
$else\ if\ a > b\ then\ nzd''\ (a - b)\ b$
$else\ if\ b > a\ then\ nzd''\ a\ (b - a)$
$else\ a)$
by *pat-completeness auto*
termination
apply (*relation measure* (λ (a, b). $a + b$))
apply *auto*
done

ili kraće `by (relation "measure (\<lambda> (a, b). a + b)", auto)`

Sada možemo pokušati da vratimo ključnu reč *fun* i vidimo da Isabelle/HOL sada uspeva sam da potvrdi da je ova definicija dobra (što nam potvrđuje ispisana poruka: "Found termination order") tako što je pronašao neku relaciju poretka nad ovim argumentima koja se tokom rekurzivnih poziva smanjuje i koja je dobro zasnovana odnosno nema beskonačnih lanaca (u ovom slučaju, leksikografsko poređenje uređenih parova).

```
fun nzd' :: nat ⇒ nat ⇒ nat where
  nzd' a b =
    (if a = 0 then b
     else if b = 0 then a
     else if a > b then nzd' (a − b) b
     else if b > a then nzd' a (b − a)
     else a)
```

Ova definicija bi se mogla zapisati i kraće bez poslednje else grane, tako što bi se ona uvukla u slučaj $a \geq b$.

Zadatak 7.64. Dokažati da je naša funkcija dobro definisana, odnosno da izračunava nzd tako što ćemo pokazati da je ekvivalentna sa bibliotečkom funkcijom gcd.

Kao i u prethodnom primeru koristi se specifično pravilo indukcije koje je dodeljeno ovoj funkciji i navode se dva argumenta. Ovde možemo prikazati kako izgled same definicije funkcije utiče na izgled dokaza koji se dobija u Isabelle/HOL. Ako bismo koristili gore navedeni oblik koji koristi samo if-then-else konstrukciju dobili bismo dokaz koji ima samo jednu granu i onda je na nama da u okviru te grane raspišemo sve moguće slučajeve koji su navedeni u okviru if-then-else naredbi (kao što smo videli u prethodnom primeru) i dobijamo naredni dokaz. Prvo izbacujemo rekurzivnu funkciju iz simplifikatora i dodaćemo je u slučajevima kada nam je potrebna:

```
declare nzd'.simps [simp del]

lemma nzd'-gcd:
  shows nzd' a b = gcd a b
proof (induction a b rule: nzd'.induct)
  case (1 a b)
  show ?case
  proof (cases b = 0)
    case True
    thus ?thesis
      by (simp add: nzd'.simps)
  next
```

```
    case False
    show ?thesis
    proof (cases a = 0)
      case True
      thus ?thesis
        using ⟨b ≠ 0⟩
        by (simp add: nzd'.simps)
    next
      case False
      show ?thesis
      proof (cases a > b)
        case True
        thus ?thesis
          using ⟨a ≠ 0⟩ ⟨b ≠ 0⟩ 1(1)
          by (simp add: nzd'.simps[of a b] gcd-diff1-nat)
      next
        case False
        show ?thesis
        proof (cases a < b)
          case True
          thus ?thesis
            using ⟨a ≠ 0⟩ ⟨b ≠ 0⟩ ⟨¬ (a > b)⟩ 1(2)
            using nzd'.simps[of a b]
            by (metis gcd.commute gcd-diff1-nat less-imp-le)
        next
          case False
          thus ?thesis
            using ⟨a ≠ 0⟩ ⟨b ≠ 0⟩ ⟨¬ (a > b)⟩
            using nzd'.simps[of a b]
            by simp
        qed
      qed
    qed
  qed
qed
```

Alternativa je da promenimo izgled same definicije i da je zapišemo na sledeći način:

```
fun nzd :: nat ⇒ nat ⇒ nat where
nzd 0 b = b |
nzd a 0 = a |
nzd a b = (if a ≥ b then nzd (a − b) b
           else nzd a (b − a))
```

sada dobijamo tri slučaja u automatski generisanom dokazu koji odgovaraju ovim trima slučajevima iz definicije. Za razliku od prethodnog oblika definicije i dokaza koji smo tada kreirali (kada smo morali da eksplicitno navodimo da prethodni slučajevi grananja nisu ispunjeni, konkretno da a i b nisu jednaki nuli), ovako definisani šabloni ne zahtevaju to. Ovako kreirana funkcija automatski vuče kroz dokaz informaciju da svi prethodni slučajevi nisu ispunjeni.

Da bi se izbeglo "zaglavljivanje" simplifikatora, ponovo rekurzivnu definiciju izbacujemo iz procesa automatske simplifikacije:

declare *nzd.simps* [*simp del*]

Slično kao i ranije glavna lema *nzd-gcd* će se svesti na matematičku formulu koju treba dokazati i za to možemo upotrebiti sledgehammer.

lemma *nzd-gcd*:
  shows *nzd a b* = *gcd a b*
proof (*induction a b rule*: *nzd.induct*)
  case (*1 b*)
  then show *?case*
    by (*simp add*: *nzd.simps*)
next
  case (*2 v*)
  then show *?case*
    by (*simp add*: *nzd.simps*)
next
  case (*3 a′ b′*) — promenićemo imena na a' i b'
— ovo je mesto gde ćemo se pozivati na rekurzivne pozive i prvo ćemo pokušati automatski i dobićemo matematičku formulu koju treba dokazati i za to možemo upotrebiti sledgehammer ili izdvojiti dve matematičke formule kao nezavisne leme.
  then show *?case*
    apply (*auto simp add*: *nzd.simps*)
— sledgehammer
    using *gcd-diff1-nat* **apply** *force*
 — sledgehammer
    by (*metis Suc-le-mono diff-Suc-Suc gcd.commute gcd-diff1-nat nat-le-linear*)
qed

Drugi način na koji možemo zapisati dokaz ove leme jeste da izdvojimo leme koje se tiču čisto matematičkih svojstava najvećeg zajedničkog delioca:

lemma *gcd-math1*:
  fixes *a b* :: *nat*
  assumes *a* ≥ *b*
  shows *gcd a b* = *gcd* (*a* − *b*) *b*

```
  using assms
  by (simp add: gcd-diff1-nat)

lemma gcd-math2:
  fixes a b :: nat
  assumes a < b
  shows gcd a b = gcd a (b − a)
  using assms
  by (metis gcd.commute gcd-diff1-nat less-imp-le)

lemma nzd-gcd-shorter:
  shows nzd a b = gcd a b
proof (induction a b rule: nzd.induct)
  case (1 b)
  then show ?case
    by (simp add: nzd.simps)
next
  case (2 v)
  then show ?case
    by (simp add: nzd.simps)
next
  case (3 a' b')
  then show ?case
    by (auto simp add: nzd.simps gcd-math1 gcd-math2)
qed
```

Ili najkraće zapisano:

```
lemma nzd-gcd-shortest:
  shows nzd a b = gcd a b
 by (induction a b rule: nzd.induct, auto simp add: nzd.simps gcd-math1 gcd-math2)
```

Zadatak 7.65. Definisati efikasniju verziju Euklidovog algoritma koja koristi celobrojno deljenje i dokazati da je ekvivalentna bibliotečkoj funkciji.

```
fun nzd-ef :: nat ⇒ nat ⇒ nat where
  nzd-ef a b =
    (if b = 0 then a
     else nzd-ef b (a mod b))

value nzd-ef 120 48

declare nzd-ef.simps[simp del]
```

Prvo ćemo pokušati automatski da dokažemo pa krećemo od indukcije za novu definiciju. Nakon toga pokušavamo samo auto i vidimo da smo stigli do čisto matematičkog tvrđenja pa sada možemo da pokušamo sa sledgehammer-om:

lemma *nzd-ef-gcd*:
nzd-ef a b = gcd a b
apply (*induction a b rule*: *nzd-ef.induct*)
apply (*auto simp add*: *nzd-ef.simps*)
using *gcd-red-nat* **by** *auto*

Ovaj dokaz bi se kraće mogao zapisati na naredni način:

```
lemma nzd_ef_gcd:
  "nzd_ef a b = gcd a b"
  using gcd_red_nat
  by (induction a b rule: nzd_ef.induct, auto simp add: nzd_ef.simps)
```

Isar dokaz:

lemma *nzd-ef-gcd-isar*:
nzd-ef a b = gcd a b
proof (*induction a b rule*: *nzd-ef.induct*)
case (*1 a b*)
show *?case*
proof (*cases b = 0*)
case *True*
thus *?thesis*
by (*simp add*: *nzd-ef.simps*)
next
case *False*
thus *?thesis*
using *1 nzd-ef.simps*[*of a b*]
using *gcd-red-nat*
by *auto*
qed
qed

U realnim situacijama nemamo na raspolaganju bibliotečku funkciju. Jer ako bismo je imali, onda ne bi imali potrebe da definišemo svoju funkciju. Tada direktno dokazujemo svojstva naše funkcije. Pokušaćemo sada to da uradimo, npr. dokažimo direktno da *nzd-ef* izračunava najveći zajednički delioc dva broja.

Specifikacija: NZD je broj koji deli i prvi i drugi argument i ujedno je i najveći broj sa tim svojstvom. Ovo tvrđenje ćemo formulisati kao dve odvojene leme.

```
lemma nzd-ef-deli-oba:
  nzd-ef a b dvd a ∧ nzd-ef a b dvd b
proof (induction a b rule: nzd-ef.induct)
  case (1 a b)
  show ?case
  proof (cases b = 0)
    case True
    thus ?thesis
      by (simp add: nzd-ef.simps)
  next
    case False
    thus ?thesis
      using nzd-ef.simps[of a b] 1
      — sledgehammer
      using dvd-mod-iff
      by auto
  qed
qed
```

Sada ćemo formulisati drugu lemu koja glasi: Ako bilo koji broj deli oba argumenta, onda on mora da deli i rezultat (rezultat je, dakle, najveći u relaciji deljivosti dvd). Nezavisno možemo pokazati i da je nzd najveći i u smislu relacije manje jednako.

```
lemma delioc-deli-nzd-ef:
  assumes c dvd a c dvd b
  shows c dvd nzd-ef a b
  using assms
proof (induction a b rule: nzd-ef.induct)
  case (1 a b)
  show ?case
  proof (cases b = 0)
    case True
    then show ?thesis
      using 1(2)
      by (simp add: nzd-ef.simps)
  next
    case False
    then show ?thesis
      using nzd-ef.simps[of a b]
      using 1
      using dvd-mod-iff
      by auto
  qed
```

qed

7.4.3 Brzo sortiranje

Zadatak 7.66. Definišimo i dokažimo korektnost funkcije brzog sortiranja.

Ukratko opisano, quick-sort se izvodi tako što prvi element liste uzimamo za pivot, pa:

- izvajamo i sortiramo manje ili jednake od pivota,
- zatim na rezultat nadovezujemo pivot,
- izdvajamo i sortiramo veće ili jednake od pivota.

Bibliotečka funkcija filter izdvaja elemente koji zadovoljavaju dati uslov. Uslov se često zadaje u obliku lambda izraza (anonimne funkcije).

value *filter* (λ *x*. *x* < *5*) [*1*::*nat*, *3*, *8*, *4*, *2*, *5*, *6*, *3*]
value *filter* (λ *x*. *x* $\geq$ *5*) [*1*::*nat*, *3*, *8*, *4*, *2*, *5*, *6*, *3*]

fun *qsort* :: *nat list* $\Rightarrow$ *nat list* **where**
qsort [] = [] |
qsort (*x* # *xs*) =
  qsort (*filter* (λ *y*. *y* $\leq$ *x*) *xs*) @ [*x*] @ *qsort* (*filter* (λ *y*. *y* > *x*) *xs*)

Primetite da je prva podlista dobijena od elemenata koji su manji ili jednaki početnom elementu liste x, ali pošto se filtrira rep liste nećemo dobiti ponavljanje elementa x.

Dokazati da ovakvo filtriranje skupa elemenata liste čuva skup elemenata i multiskup elemenata početne liste:

lemma *mset-filter-leq-filter-gt*:
  fixes *xs* :: *nat list*
  shows *mset* (*filter* (λ*y*. *y* $\leq$ *x*) *xs*) + *mset* (*filter* (λ *y*. *y* > *x*) *xs*) =
    mset xs
  by (*induction xs*) *auto*

lemma *set-filter-leq-filter-gt*:
  fixes *xs* :: *nat list*
  shows *set* (*filter* (λ*y*. *y* $\leq$ *x*) *xs*) $\cup$ *set* (*filter* (λ *y*. *y* > *x*) *xs*) =
    set xs

by *auto*

Dokazati da se multiskup elemenata originalne liste ne menja nakon sortiranja:

lemma *mset-qsort*:
mset (*qsort xs*) = *mset xs*
by (*induction xs rule*: *qsort.induct*) (*auto simp add*: *not-less*)

Samim tim se ne menja ni skup elemenata:

lemma *set-qsort*:
set (*qsort xs*) = *set xs*
by (*metis mset-qsort set-mset-mset*)

Dokazati da je dobijena lista nakon sortiranja sortirana:

Napomena: u narednom dokazu, obratite pažnju na to da su sve leme koje se koriste u dokazu već ranije formulisane i dokazane. Ako to ne bi bio slučaj, na nama je da prepoznamo koji format treba da imaju nove leme, da ih formulišemo i dokažemo.

Za vežbu možete pokušati da u nezavisnom dokumentu izdvojite samo definicije i glavne leme i da vidite da li možete da prepoznate šta je sve potrebno dodatno formulisati.

lemma *qsort-sorted*:
sorted (*qsort xs*)
apply (*induction xs rule*: *qsort.induct*)
apply *auto*
apply (*auto simp add*: *sorted-append*)
apply (*auto simp add*: *set-qsort*)
done

7.4.4 Sortiranje objedinjavanjem

Zadatak 7.67. Definisati i analizirati funkciju koja spaja dve sortirane liste tako da formira novu sortiranu listu.

fun *spoji* :: *nat list* ⇒ *nat list* ⇒ *nat list* **where**
spoji [] *ys* = *ys* |
spoji xs [] = *xs* |
spoji (*x* # *xs*) (*y* # *ys*) =
(*if x* ≤ *y then*
x # *spoji xs* (*y* # *ys*)
else
y # *spoji* (*x* # *xs*) *ys*)

Pokazati da je multiskup (i skup) liste dobijene objedinjavanjem jednak uniji multiskupova (odnosno skupova) originalnih listi. Ove leme bismo mogli dodati u simplifikator (ključnom reči simp nakon imena leme) ali nećemo iskoristiti tu opciju sada da bismo videli da su neophodne za dokazivanje glavne leme.

Napomena: kada radite na manjem skupu lema u jednoj teoriji slobodno možete dodati leme ovog tipa u simplifikator, ali u opštem slučaju treba biti obazriv pošto previše lema može opteretiti simplifikator do te mere da postane neupotrebljiv.

lemma *mset-spoji*:
shows *mset* (*spoji xs ys*) = *mset xs* + *mset ys*
by (*induction xs ys rule*: *spoji.induct*, *auto*)

lemma *set-spoji*:
shows *set* (*spoji xs ys*) = *set xs* ∪ *set ys*
apply (*induction xs ys rule*: *spoji.induct*)
apply *auto*
done

Dokazati da se funkcijom *spoji* od dve sortirane liste dobija sortirana lista:

lemma *sorted-spoji*:
assumes *sorted xs sorted ys*
shows *sorted* (*spoji xs ys*)
using *assms*
apply (*induction xs ys rule*: *spoji.induct*)
apply *auto* — vidimo da je automatski dokazivač došao do skupova, javlja se izraz oblika *set* (*spoji* - -) pa dodajemo lemu *set-spoji*. Napomena, da ova lema nije već bila formulisana morali bismo sami da prepoznamo da je ovakva lema potrebna i formulisali bismo je i dokazali.
apply (*auto simp add*: *set-spoji*)
done

Zadatak 7.68. Napisati funkciju koja sortira listu objedinjavanjem i dokazati da je dobro definisana.

Koristićemo ugrađene funkcije *take* i *drop* koje uzimaju odnosno izbacuju prvih n elemenata liste.

fun *mergesort* :: *nat list* ⇒ *nat list* **where**
mergesort [] = [] |
mergesort [*x*] = [*x*] |
mergesort xs =
(*let n* = *length xs div 2*

in spoji (*mergesort* (*take n xs*)) (*mergesort* (*drop n xs*)))

value *length* [*3*, *5*, *1*, *4*, *8*, *2*::*nat*] *div 2*
value *take 3* [*3*, *5*, *1*, *4*, *8*, *2*::*nat*]
value *drop 3* [*3*, *5*, *1*, *4*, *8*, *2*::*nat*]
value *mergesort* [*3*, *5*, *1*, *4*, *8*, *2*]

Pokazati da je lista dobijena funkcijom mergesort sortirana:

lemma *sorted-mergesort*:
shows *sorted* (*mergesort xs*)
apply (*induction xs rule*: *mergesort.induct*)
apply *auto* — vidimo da je dokazivač stao kod oblika *sorted* (*spoji* - -) pa se pozivamo na lemu *sorted-spoji*
apply (*auto simp add*: *sorted-spoji*)
done

Dokazati da je skup elemenata i multiskup elemenata ostao nepromenjen nakon poziva funkcije mergesort:

lemma *mset-take-drop*:
shows *mset* (*take n xs*) + *mset* (*drop n xs*) = *mset xs*
— sledgehammer
by (*metis append-take-drop-id mset-append*)

lemma *mset-mergesort*:
shows *mset* (*mergesort xs*) = *mset xs*
apply (*induction xs rule*: *mergesort.induct*)
apply *auto*
— možemo sami, korak po korak doći do ovog zaključka ili upotrebiti sledgehammer
— primetimo da u zaključku dokazivač ima izraz oblika *mset* (*spoji* - -) pa pokušavamo da dodamo lemu tog oblika (koju smo već dokazali)
apply (*auto simp add*: *mset-spoji*)
— sada možemo da primetimo da u zaključku imamo izraz oblika *mset* (*take* - -) + *mset* (*drop* - -) pa sada formulišemo novu lemu tog oblika koju ćemo nazvati *mset-take-drop*.
apply (*auto simp add*: *mset-take-drop*)
done

Slično za set:

lemma *set-take-drop*:
shows *set* (*take n xs*) ∪ *set* (*drop n xs*) = *set xs*
— sledgehammer
by (*metis append-take-drop-id set-append*)

lemma *set-mergesort*:
shows *set* (*mergesort xs*) = *set xs*
apply (*induction xs rule*: *mergesort.induct*)
apply *auto*
apply (*metis Un-iff in-set-dropD in-set-takeD list.simps*(*15*) *set-ConsD set-spoji*)
apply (*auto simp add*: *set-spoji*)
apply (*metis drop-Cons′ list.set-intros*(*1*) *take-Cons′*)
— sledgehammer sam ne uspeva pa dodajemo novu lemu, slično kao za mset i onda ponovo sledgehammer
by (*metis Un-iff insertI2 list.simps*(*15*) *set-take-drop*)

7.4.5 Deljenje liste

Definisaćemo funkciju mergesort sada na malo drugačiji način:

Zadatak 7.69. Definisati funkciju koja deli listu na dve liste približno jednake dužine.

Ovakva funkcija vraća uređeni par listi. U njenoj implementaciji koristićemo funkcije *take* i *drop* koje izdvajaju, odnosno izbacuju, prvih k elemenata liste.

value *length* [*3*::*nat*, *1*, *4*, *2*, *5*] — duzina liste
value *take 2* [*3*::*nat*, *1*, *4*, *2*, *5*] — uzimanje prvih k elemenata liste
value *drop 2* [*3*::*nat*, *1*, *4*, *2*, *5*] — izbacivanje prvih k elemenata liste

definition *split* :: *′a list* ⇒ *′a list* × *′a list* **where**
split xs = (*let k* = *length xs div 2 in* (*take k xs*, *drop k xs*))

Dokazati da nakon poziva funkcije split, dobijamo kraće liste:

lemma *length-split*:
assumes *length xs* > *1* (*ys*, *zs*) = *split xs*
shows *length ys* < *length xs length zs* < *length xs*
using *assms*
unfolding *split-def Let-def*
by *auto*

Dokazati da je multiskup elemenata podeljen u dve rezultujuće liste:

lemma *mset-split*:
assumes (*ys*, *zs*) = *split xs*
shows *mset xs* = *mset ys* + *mset zs*
using *assms*
unfolding *split-def Let-def*

by (*simp add*: *union-code*)

Da bi se moglo nešto zaključivati o vrednosti generalno-rekurzivne funkcije, potrebno je dokazati da se ona zaustavlja. Kada se upotrebi ključna reč *fun*, sistem Isabelle/HOL automatski pokušava da dokaže zaustavljanje. To nekada uspe, a nekada ne. Kada ne uspe, onda je potrebno da korisnik sam dokaže zaustavljanje i tada koristi kombinaciju function-termination i možemo koristiti sledgehammer za dokazivanje zaustavljanja.

Potrebno je ponovo definisati funkciju f i koristiti naredbu oblika

```
relation "measure f"
```

kojom ćemo dokazati da se u svakom koraku rekurzije vrednost funkcije f smanjuje. U našem primeru tvrdimo da se kroz rekuzivne pozive smanjuje dužina liste. Nakon toga koristimo sledgehammer da bismo to i dokazali.

function (*sequential*) *mergesort-split* :: *nat list* ⇒ *nat list* **where**
mergesort-split [] = [] |
mergesort-split [*x*] = [*x*] |
mergesort-split xs =
  (*let* (*ys*, *zs*) = *split xs*
    in spoji (*mergesort-split ys*) (*mergesort-split zs*))
 by *pat-completeness auto*
termination
 apply (*relation measure* (λ *xs*. *length xs*))
  apply *simp*
 apply (*metis One-nat-def in-measure length-Cons lessI linorder-not-less list.size*(*4*) *length-split*(*1*) *trans-le-add2*)
 apply (*metis One-nat-def impossible-Cons in-measure linorder-not-less list.size*(*4*) *length-split*(*2*) *trans-le-add2*)
 done

Funkcija je definisana preko tri "šablona" (engl. pattern). Opcija `sequential` koja se navodi u zagradama, navodi se u kombinaciji sa ključnom rečju `function` , i obezbeđuje da se šabloni ispituju jedan po jedan (tj. da se tekući šablon primenjuje samo ako prethodni nisu uklopljeni - kao što se podrazumeva kada se koristi `fun`). Ovo je podrazumevano ponašanje u većini funkcionalnih jezika (npr. u Haskell-u), pa ćemo stoga praktično uvek navoditi tu opciju.

U narednom dokazu koristićemo naredbu `apply (simp split: prod.split)`. Pošto funkcija *mergesort-split* prvo kreira uređeni par *let* (*ys*, *zs*) = *split xs*, potrebno nam je da upotrebimo *split* pravilo *prod.split* koje će izdvojiti elemente odgovarajućeg uređenog para.

Napomena: ovde se *split* koristi kao ključna reč i nema veze sa imenom funkcije split koje smo ranije koristili.

```
lemma mset-mergesort-split:
  mset (mergesort-split xs) = mset xs
proof (induction xs rule: mergesort-split.induct)
  case 1
  then show ?case
    by simp
next
  case (2 x)
  then show ?case
    by simp
next
  case (3 x0 x1 xs)
  then show ?case
    apply (simp split: prod.split)
— sledgehammer
    by (metis mset.simps(2) mset-spoji mset-split)
— ili kraće:
— by (simp split: prod.split) (metis mset.simps(2) mset-spoji mset-split)
qed

lemma sorted-mergesort-split:
  sorted (mergesort-split xs)
  apply (induction xs rule: mergesort-split.induct)
    apply (auto split: prod.split)
— sledgehammer
  by (simp add: sorted-spoji)
```

Napomena: odavde pa nadalje se nalazi materijal za vežbanje. Pokušajte sami da formulišete sve definicije i leme ili makar da napišete dokaze lema nakon što vidite njihovu formulaciju.

7.4.6 Sortiranje umetanjem

Napisati funkciju koja ubacuje element u sortiranu listu prirodnih brojeva tako da ona ostane i dalje sortirana.

```
primrec ubaci :: nat ⇒ nat list ⇒ nat list where
ubaci a [] = [a] |
ubaci a (x # xs) = (if a ≤ x then a # x # xs else x # ubaci a xs)

value ubaci 5 [1,3,4,9]
```

Napisati funkciju koja sortira listu umetanjem, prvo uz pomoć rekurzije a onda uz pomoć funkcije fold.

primrec *insertion-sort-rek* :: *nat list* ⇒ *nat list* **where**
insertion-sort-rek [] = [] |
insertion-sort-rek (*x* # *xs*) = *ubaci x* (*insertion-sort-rek xs*)

Nakon što smo definisali funkciju ubaci, možemo iskoristiti funkciju fold da ubacimo sve elemente liste u praznu listu.

definition *insertion-sort-fold* :: *nat list* ⇒ *nat list* **where**
insertion-sort-fold xs = *fold ubaci xs* []

value *insertion-sort-fold* [*3*,*1*,*5*,*4*]
value *insertion-sort-fold* []

term *sorted*
value *sorted* [*3*,*1*,*2*::*nat*]

Dokazati da su funkcije *ubaci*, *insertion-sort-fold*, *insertion-sort-rek* dobro definisane u odnosu na skup:

lemma [*simp*]:
set (*ubaci a xs*) = {*a*} ∪ *set xs*
by (*induction xs*) *auto*

lemma *fold f* [*a*,*b*,*c*] *x* = *f c* (*f b* (*f a x*)) **by** *simp*
lemma *foldr f* [*a*,*b*,*c*] *x* = *f a* (*f b* (*f c x*)) **by** *simp*
lemma *foldl f x* [*a*,*b*,*c*] = *f* (*f* (*f x a*) *b*) *c* **by** *simp*

Koristi se desni fold pa nam treba indukcija sa desna, odnosno indukcija kada se novi element dodaje na kraj liste ⟦*P* []; ⋀*x xs*. *P xs* ⟹ *P* (*xs* @ [*x*])⟧ ⟹ *P xs*.
https://www.cl.cam.ac.uk/research/hvg/Isabelle/dist/library/Doc/How_to_Prove_it/How_to_Prove_it.html

thm *rev-induct*

lemma *set* (*insertion-sort-fold xs*) = *set xs*
unfolding *insertion-sort-fold-def*
by (*induction xs rule*: *rev-induct*) *auto*

lemma *set* (*insertion-sort-rek xs*) = *set xs*
unfolding *insertion-sort-rek-def*
by (*induction xs*) *auto*

Dokazati da su funkcije *ubaci*, *insertion-sort-fold*, *insertion-sort-rek* dobro definisane u odnosu na multiskup:

Drugi način da ubacimo element u multiskup (osim $\{\#a\#\}$ + *mset xs*) je korišćenjem funkcije *add-mset*:

term *mset*
term *add-mset*

lemma [*simp*]:
mset (*ubaci a xs*) = *add-mset a* (*mset xs*)
by (*induction xs*) *auto*

lemma [*simp*]:
mset (*insertion-sort-fold xs*) = *mset xs*
unfolding *insertion-sort-fold-def*
by (*induction xs rule*: *rev-induct*) *auto*

lemma [*simp*]:
mset (*insertion-sort-rek xs*) = *mset xs*
unfolding *insertion-sort-rek-def*
by (*induction xs*) *auto*

Dokazati da funkcije *ubaci*, *insertion-sort-fold*, *insertion-sort-rek* generišu sortirane liste:

lemma [*simp*]:
assumes *sorted xs*
shows *sorted* (*ubaci a xs*)
using *assms*
by (*induction xs*) *auto*

lemma
sorted (*insertion-sort-fold xs*)
by (*induction xs rule*: *rev-induct*) (*auto simp add*: *insertion-sort-fold-def*)

lemma
sorted (*insertion-sort-rek xs*)
by (*induction xs*) *auto*

7.4.7 Celobrojno deljenje

Opštom rekurzijom definisati funkciju div2 koja izračunava celobrojnu vrednost n/2 i dokazati da je dobro definisana.

fun *div2* :: *nat* ⇒ *nat* **where**

div2 0 = 0 |
div2 (Suc 0) = 0 |
div2 (Suc(Suc n)) = Suc (div2 n)

thm *div2.induct*

lemma *div2 n = n div 2*
apply (*induction n rule*: *div2.induct*)
apply *auto*
done

Ranije smo već definisali sabiranje nad prirodnim brojevima na sledeći način:

primrec *add* :: *nat* ⇒ *nat* ⇒ *nat* **where**
add 0 n = n |
add (Suc m) n = Suc (add m n)

Napisati funkciju itadd koristeći repnu rekurziju, odnosno napisati rekurzivni poziv tako da funkcija itadd poziva samu sebe direktno: *itadd (Suc m) n = itadd*.... Dokazati da je ekvivalentna operaciji sabiranja koju smo malopre uveli.

primrec *itadd* :: *nat* ⇒ *nat* ⇒ *nat* **where**
itadd 0 n = n |
itadd (Suc m) n = itadd m (Suc n)

Pomoćna lema:

lemma *itadd-Suc*:
shows *itadd m (Suc n) = Suc (itadd m n)*
apply (*induction m arbitrary*: *n*)
— posto funkcija itadd menja drugi argument, moramo dodati ključnu reč arbitrary
apply *auto*
done

lemma *itadd m n = add m n*
apply (*induction m*)
apply *auto*
— vidimo da je dokazivač stao kod tvrđenja oblika *itadd m (Suc n) = Suc (add m n)* pa formulišemo pomoćnu lemu *itadd-Suc*koja odgovara tom tvrđenju
apply (*auto simp add*: *itadd-Suc*)
done

Ili kraće: `by (induction m) (auto simp add: itadd_Suc)`

7.4.8 Pretraga za parom brojeva

Napisati funkciju koja proverava da li u sortiranoj listi prirodnih brojeva postoji par brojeva čiji zbir je jednak traženom broju s. Funkcija vraća True ili False.

Pošto se u rekurzivnim pozivima ove funkcije menjaju indeksi granica liste (odnosno dela liste koji pretražujemo) funkcija mora da ima dva početna parametra koji će odgovarati levoj i desnoj granici - lo i hi.

U ovoj funkciji koristimo operator $xs!i$ kojim pristupamo elementu liste xs koji se nalazi na poziciji i.

```
function par :: nat ⇒ nat ⇒ nat list ⇒ nat ⇒ bool where
  par lo hi xs s = (
    if lo ≥ hi then
       False
    else if xs ! lo + xs ! hi < s then
       par (lo + 1) hi xs s
    else if xs ! lo + xs ! hi > s then
       par lo (hi − 1) xs s
    else
       True)
  by pat-completeness auto
termination by (relation measure (λ (lo, hi, xs, x). hi − lo)) auto

declare par.simps [simp del]
```

Dokazati da ako je lista sortirana, funkcija par vraća True ako i samo ako u listi postoje dva prirodna broja na pozicijama i i j čiji zbir je jednak traženom broju s.

Dokaz počinjemo indukcijom, samo moramo voditi računa da navedemo sve parametre funkcije par onim redom kojim se pojavljuju u definiciji. Nakon toga dokaz granamo onim redom kojim se pojavljuju grananja u definiciji. Počinjemo sa grananjem $lo \geq hi$, pa uvodimo redom sva grananja koja se javljaju u definiciji.

Za vežbu: pokušajte sami da napišete ovaj dokaz.

```
lemma sorted-par:
  assumes sorted xs 0 ≤ lo hi < length xs
  shows par lo hi xs s ⟷
        (∃ i j. lo ≤ i ∧ i < j ∧ j ≤ hi ∧ xs ! i + xs ! j = s)
  using assms
proof (induction lo hi xs s rule: par.induct)
  case (1 lo hi xs s)
```

```
show ?case
proof (cases lo ≥ hi)
  case True
  thus ?thesis
    by (simp add: par.simps)
next
  case False
  show ?thesis
  proof (cases xs ! lo + xs ! hi < s)
    case True
    have (∃ i≥Suc lo. ∃ j>i. j ≤ hi ∧ xs ! i + xs ! j = s) ⟷
         (∃ i≥lo. ∃ j>i. j ≤ hi ∧ xs ! i + xs ! j = s) (is ?lhs ⟷ ?rhs)
    proof
      assume ?lhs
      then obtain i j where ∗: Suc lo ≤ i i < j j ≤ hi xs ! i + xs ! j = s
        by auto
      hence lo ≤ i
        by simp
      thus ?rhs
        using ∗
        by blast
    next
      assume ?rhs
      then obtain i j where ∗: lo ≤ i i < j j ≤ hi xs ! i + xs ! j = s
        by auto
      show ?lhs
      proof (cases Suc lo ≤ i)
        case True
        thus ?thesis
          using ∗
          by blast
      next
        case False
        hence i = lo
          using ∗
          by simp
        hence xs ! lo + xs ! j = s
          using ∗
          by simp
        hence xs ! lo + xs ! hi ≥ s
          using ⟨sorted xs⟩ ⟨j ≤ hi⟩ ⟨hi < length xs⟩
          using sorted-nth-mono[of xs j hi]
          by auto
```

```
      hence False
        using ⟨xs ! lo + xs ! hi < s⟩
        by simp
      thus ?thesis
        by simp
    qed
  qed
  thus ?thesis
    using ⟨¬ lo ≥ hi⟩ par.simps[of lo hi xs s]
    using ⟨xs ! lo + xs ! hi < s⟩ 1(1) 1(3−5)
    by simp
next
  case False
  show ?thesis
  proof (cases xs ! lo + xs ! hi > s)
    case True
    have (∃ i≥lo. ∃ j>i. j ≤ hi − 1 ∧ xs ! i + xs ! j = s) ⟷
          (∃ i≥lo. ∃ j>i. j ≤ hi ∧ xs ! i + xs ! j = s) (is ?lhs ⟷ ?rhs)
    proof
      assume ?lhs
      then obtain i j where ∗: lo ≤ i i < j j ≤ hi − 1 xs ! i + xs ! j = s
        by auto
      hence j ≤ hi
        by simp
      thus ?rhs
        using ∗
        by blast
    next
      assume ?rhs
      then obtain i j where ∗: lo ≤ i i < j j ≤ hi xs ! i + xs ! j = s
        by auto
      show ?lhs
      proof (cases j ≤ hi − 1)
        case True
        thus ?thesis
          using ∗
          by blast
      next
        case False
        hence j = hi
          using ∗
          by simp
        hence xs ! i + xs ! hi = s
```

```
          using ∗
          by simp
        hence xs ! lo + xs ! hi ≤ s
          using ⟨sorted xs⟩ ∗ ⟨hi < length xs⟩
          using sorted-nth-mono[of xs lo i]
          by simp
        hence False
          using ⟨xs ! lo + xs ! hi > s⟩
          by simp
        thus ?thesis
          by simp
      qed
    qed
    thus ?thesis
      using ⟨¬ lo ≥ hi⟩ par.simps[of lo hi xs s]
      using ⟨xs ! lo + xs ! hi > s⟩ 1(2) 1(3−5)
      by simp
  next
    case False
    thus ?thesis
      using par.simps[of lo hi xs s] ⟨¬ lo ≥ hi⟩
      using ⟨¬ xs ! lo + xs ! hi < s⟩ ⟨¬ xs ! lo + xs ! hi > s⟩
      by auto (rule-tac x=lo in exI, simp, rule-tac x=hi in exI, simp)
  qed
 qed
qed
```

7.4.9 Binarna pretraga

Napisati funkciju koja izvodi binarnu pretragu nad sortiranom listom.

Prvo ćemo definisati pomoćnu funkciju bsearch' koja će imati dodatna dva argumenta koji će određivati opseg liste koji pretražujemo.

Primetimo da naredna funkcija mora biti deklarisana sa celim brojevima, zato što ovakva definicija dodeljuje promenljivoj mid tip celog broja, a ta promenljiva se prosleđuje narednim rekurzivnim pozivima (pogledajte šta se dešava kada promenite tip u funkciji sa int na nat).

```
function bsearch′ :: int ⇒ int ⇒ nat list ⇒ nat ⇒ bool where
 bsearch′ lo hi xs x =
  (if lo > hi then
     False
   else
```

```
      let mid = lo + (hi − lo) div 2
       in if x > xs ! nat mid then
            bsearch′ (mid+1) hi xs x
          else if x < xs ! nat mid then
            bsearch′ lo (mid−1) xs x
          else
            True
  )
  by pat-completeness auto
termination
by (relation measure (λ (lo, hi, xs, x). nat (hi − lo + 1))) auto

declare bsearch′.simps [simp del]
```

Sada možemo definisati funkciju bsearch koja prima samo sortiranu listu i broj koji se traži:

```
definition bsearch :: nat list ⇒ nat ⇒ bool where
  bsearch xs x = bsearch′ 0 (int (length xs) − 1) xs x
```

Dokazati da je funkcija bsearch dobro definisana.

Razmotrimo šta to znači u ovom slučaju. Lista koje se pretražuje mora biti sortirana, i funkcija bsearch će vratiti vrednost True ako i samo ako se traženi element nalazi u listi. Proveru da li se element nalazi u listi izvodimo uz pomoć funkcije set i dobijamo narednu formulaciju:

```
lemma sorted_bsearch:
  assumes "sorted xs"
  shows "bsearch xs x \<longleftrightarrow> x \<in> set xs"
```

Pošto funkcija bsearch poziva funkciju bsearch' koja radi nad segmentima liste, primećujemo da nam odgovara da uvedemo pojam segmenta liste - elementi liste koji se nalaze između indeksa lo i hi na sledeći način. Koristićemo lambda funkciju i operaciju map:

```
definition segment where
  segment lo hi xs = map (λ i. xs ! (nat i)) [lo..hi]
```

Kada budemo radili sa listom, krećemo od cele liste tako da je na početku segment određen pozicijom 0 i poslednjim elementom liste koji je na poziciji *length xs − 1*. Ovo ćemo dokazati kao izdvojenu lemu i nazvaćemo je *ceo-segment*.

U narednom dokazu koristićemo lemu *map-nth* koja glasi: *map* ((!) *xs*) [*0..<length xs*] = *xs* i tvrdi da funkcija ! može biti korišćena da mapiramo

sve elemente liste kada prođemo kroz sve indekse od prvog do poslednjeg elementa.

thm *map-nth*

lemma *ceo-segment*:
shows *segment 0 (int (length xs) − 1) xs = xs*
proof−
have *[0..<length xs] = map nat [0..int(length xs) − 1]*
by (*induction xs*, *simp-all add*: *upto-rec2*)
thus *?thesis*
using *map-nth*[*of xs*, *symmetric*]
unfolding *segment-def*
— možemo da iskoristimo sledgehammer a možemo i da primenimo samo auto i da vidimo da se pojavljuje kompozicija funkcija
by (*auto simp add*: *comp-def*)
qed

Sada ćemo ponovo dodati opštiju lemu koja će se ticati funkcije bsearch'. Ekvivalent našoj finalnoj lemi, samo primenjena na segment lo hi.

U narednom dokazu, isto kao i ranije, moramo pratiti definiciju same funkcije i u ovom slučaju u definiciji imamo naredbu `let mid = lo + (hi - lo) div 2` koju koristimo u dokazu. Nadalje granamo isto kao i u ranijim dokazima.

lemma *bsearch′-segment*:
assumes *0 ≤ lo lo ≤ hi hi < int (length xs) sorted xs*
shows *bsearch′ lo hi xs x ⟷ x ∈ set (segment lo hi xs)*
using *assms*
proof (*induction lo hi xs x rule*: *bsearch′.induct*)
case (*1 lo hi xs x*)
let *?mid = lo + (hi − lo) div 2*

have *lo ≤ ?mid ?mid ≤ hi*
using ‹*lo ≤ hi*›
by *auto*

show *?case*
proof (*cases x > xs ! (nat ?mid)*)
case *True*
hence ∗: *bsearch′ lo hi xs x = bsearch′ (?mid + 1) hi xs x*
using ‹*lo ≤ hi*› *bsearch′.simps*[*of lo hi xs x*]
by *simp*

have *x ∉ set (segment lo ?mid xs)*

```
proof (rule ccontr)
  assume ¬ ?thesis
  then obtain i where lo ≤ i i ≤ ?mid xs ! nat i = x
    by (auto simp add: segment-def)
  hence xs ! nat i > xs ! nat ?mid
    using ‹x > xs ! nat ?mid›
    by simp
  moreover
  have nat i ≤ nat ?mid
    using ‹i ≤ ?mid›
    by simp
  moreover
  have ?mid < int (length xs)
    using ‹?mid ≤ hi› ‹hi < int (length xs)›
    by simp
  hence nat ?mid < length xs
    using ‹0 ≤ lo› ‹lo ≤ ?mid›
    by simp
  ultimately
  show False
    using ‹sorted xs›
    using sorted-nth-mono[of xs nat i nat ?mid]
    by simp
qed

have [lo..hi] = [lo..?mid] @ [?mid+1..hi]
  using ‹lo ≤ ?mid› ‹?mid ≤ hi›
  using upto-split2
  by blast
hence segment lo hi xs = segment lo ?mid xs @ segment (?mid + 1) hi xs
  unfolding segment-def
  by simp
hence ∗∗: x ∈ set (segment lo hi xs) ⟷ x ∈ set (segment (?mid + 1) hi xs)
  using ‹x ∉ set (segment lo ?mid xs)›
  by simp

show ?thesis
proof (cases ?mid + 1 ≤ hi)
  case False
  have segment (?mid + 1) hi xs = []
    using False ‹0 ≤ lo› ‹lo ≤ hi›
    unfolding segment-def
    by simp
```

```
    moreover
    have bsearch' (?mid + 1) hi xs x = False
      using False bsearch'.simps[of ?mid + 1 hi xs x]
      by simp
    ultimately
    show ?thesis
      using * **
      by simp
  next
    case True
    thus ?thesis
      using 1(1)[of ?mid] ‹lo ≤ hi› ‹0 ≤ lo› ‹hi < int (length xs)› ‹sorted xs›
      using ‹x > xs ! nat ?mid› * **
      by simp
  qed
next
  case False
  show ?thesis
  proof (cases x < xs ! nat ?mid)
    case True
    hence *: bsearch' lo hi xs x = bsearch' lo (?mid - 1) xs x
      using ‹lo ≤ hi› bsearch'.simps[of lo hi xs x]
      by simp

    have x ∉ set (segment ?mid hi xs)
    proof (rule ccontr)
      assume ¬ ?thesis
      then obtain i where ?mid ≤ i i ≤ hi xs ! nat i = x
        by (auto simp add: segment-def)
      hence xs ! nat i < xs ! nat ?mid
        using ‹x < xs ! nat ?mid›
        by simp
      moreover
      have nat i ≥ nat ?mid
        using ‹i ≥ ?mid›
        by simp
      moreover
      have nat i < length xs
        using ‹0 ≤ lo› ‹lo ≤ hi› ‹hi < int (length xs)› ‹i ≤ hi›
        by simp
      ultimately
      show False
        using ‹sorted xs›
```

```
      using sorted-nth-mono[of xs nat ?mid nat i]
      by simp
  qed

  have [lo..hi] = [lo..?mid−1] @ [?mid..hi]
    using ⟨lo ≤ ?mid⟩ ⟨?mid ≤ hi⟩
    using upto-split1
    by blast
  hence segment lo hi xs = segment lo (?mid − 1) xs @ segment ?mid hi xs
    unfolding segment-def
    by simp
 hence ∗∗: x ∈ set (segment lo hi xs) ⟷ x ∈ set (segment lo (?mid − 1) xs)
    using ⟨x ∉ set (segment ?mid hi xs)⟩
    by simp

  show ?thesis
  proof (cases lo ≤ ?mid − 1)
    case False
    have segment lo (?mid − 1) xs = []
      using False ⟨0 ≤ lo⟩ ⟨lo ≤ hi⟩
      unfolding segment-def
      by simp
    moreover
    have bsearch′ lo (?mid − 1) xs x = False
      using False bsearch′.simps[of lo ?mid − 1 xs x]
      by simp
    ultimately
    show ?thesis
      using ∗ ∗∗
      by simp
  next
    case True
    thus ?thesis
      using 1(2)[of ?mid] ⟨lo ≤ hi⟩ ⟨0 ≤ lo⟩ ⟨hi < int (length xs)⟩ ⟨sorted xs⟩
      using ⟨x < xs ! nat ?mid⟩ ∗ ∗∗
      by simp
  qed
next
  case False
  have bsearch′ lo hi xs x = True
    using bsearch′.simps[of lo hi xs x]
    using ⟨¬ x > xs ! nat ?mid⟩ ⟨¬ x < xs ! nat ?mid⟩ ⟨lo ≤ hi⟩
    by simp
```

```
    moreover
    have x = xs ! nat ?mid
      using ⟨¬ x > xs ! nat ?mid⟩ ⟨¬ x < xs ! nat ?mid⟩
      by simp
    hence x ∈ set (segment lo hi xs)
      using ⟨lo ≤ ?mid⟩ ⟨?mid ≤ hi⟩
      unfolding segment-def
      by simp
    ultimately
    show ?thesis
      by simp
  qed
 qed
qed
```

Sada prelazimo na glavnu lemu. Težinu dokaza smo prebacili na prethodne leme tako da je dokaz ove leme kratak.

```
lemma sorted-bsearch:
 assumes sorted xs
 shows bsearch xs x ⟷ x ∈ set xs
proof (cases xs = [])
 case True
 thus ?thesis
   unfolding bsearch-def
   by (simp add: bsearch'.simps)
next
 case False
 hence segment 0 (int (length xs) − 1) xs = xs
   by (auto simp add: ceo-segment)
 thus ?thesis
   unfolding bsearch-def
   using assms ⟨xs ≠ []⟩
   using bsearch'-segment[of 0 int (length xs) − 1 xs x]
   by simp
qed

end
```

7.5 Računanje vrednosti prefiksno zadatog izraza i generisanje koda za stek mašinu

theory *Cas13-vezbe*
imports *HOL−Library.Mapping HOL−Library.RBT-Mapping*

begin

Tehnike korišćene za definisanje i rad sa binarnim drvetom mogu se koristiti za definisanje i izračunavanje vrednosti prefiksno zadatog izraza. U ovom poglavlju ćemo prikazati kako možemo definisati tip prefiksnog izraza nad prirodnim brojevima, sa unapred određenim skupom dozvoljenih operacija.

7.5.1 Prefiksni izraz nad konstantama

Prefiksni izraz: Prefiksno zadat izraz nad konstantama tipa *nat*, sa operacijama sabiranja, oduzimanja i množenja (koje ćemo označiti *PlusI*, *MinusI*, *PutaI*) se može definisati na sledeći način:

datatype
Izraz = *Const nat* | *PlusI Izraz Izraz* | *MinusI Izraz Izraz* | *PutaI Izraz Izraz*

term *PlusI* — operator
term *PlusI* (*Const 3*) (*Const 5*) — primer prefiksno zadatog izraza

Ova definicija veoma podseća na implementaciju koja bi mogla da bude korišćena u jeziku Haskell tako da ćemo parsiranje ovakvog izraza implementirati na sličan način kao što bi se parsirao aritmetički izraz u Haskell-u.

Vrednost izraza: Sada lako možemo definisati funkciju koja računa vrednost ovako zadatog izraza. Vrednost izraza koji se sastoji samo od konstante je upravo ta konstanta, u suprotnom se vrednost izraza računa prema operatoru koji ga gradi.

Napomena: kako radimo sa prirodnim brojevima, vrednost oduzimanja će uvek biti broj koji je veći ili jednak od 0.

primrec *vrednost* :: *Izraz* ⇒ *nat* **where**
vrednost (*Const x*) = *x* |
vrednost (*PlusI i1 i2*) = *vrednost i1* + *vrednost i2* |
vrednost (*PutaI i1 i2*) = *vrednost i1* ∗ *vrednost i2* |
vrednost (*MinusI i1 i2*) = *vrednost i1* − *vrednost i2*

value *vrednost* (*PlusI* (*Const 3*) (*Const 5*)) — rezultat je 8, primetimo da moramo koristiti zagrade oko ovako zapisanog izraza
value *vrednost* (*MinusI* (*Const 3*) (*Const 5*)) — rezultat je 0

Uz ključnu reč `definition` možemo da uvedemo nekoliko definicija:

definition *x1* :: *Izraz* **where**
x1 = *PlusI* (*Const 3*) (*Const 5*)
value *vrednost x1* — rezultat je 8

definition *x2* :: *Izraz* **where**
x2 = *PutaI* (*Const 3*)(*MinusI* (*Const 5*) (*Const 2*))
value *vrednost x2* — rezultat je 9

definition *x3* :: *Izraz* **where**
x3 = *PutaI* (*PlusI* (*Const 3*) (*Const 4*)) (*MinusI* (*Const 5*)(*Const 2*))
value *vrednost x3* — rezultat je 21

Na ovaj način smo prikazali kako možemo direktno da izračunamo vrednost prefiksno zadatog izraza. Sada ćemo pokazati drugi način za računanje ove vrednosti. Želimo da izgenerišemo program koji bi izračunavao vrednost ovako zadatog prefiksnog izraza. Prvi korak je definisanje skupa dozvoljenih operacija.

Dozvoljene operacije: Pored operacija nad prirodnim brojevima (množenje, sabiranje i oduzimanje), potrebno je i da definišemo operaciju smeštanja elementa na stek. Kako radimo sa prirodnim brojevima, stek možemo predstaviti uz pomoć liste prirodnih brojeva. Uvešćemo ga uz pomoć ključne reči `type_synonym`. Na ovaj način dobijamo samo skraćeni zapis za listu prirodnih brojeva i povećavamo čitljivost našeg koda. Interno, ovakve skraćenice se u potpunosti raspisuju.

datatype *Operacija* = *OpPuta* | *OpPlus* | *OpMinus* | *OpPush nat*

type-synonym *Stek* = *nat list*

Primena operacija: Sve operacije se izvršavaju nad stekom na uobičajeni način. Operacija smeštanja elementa na stek u stvari dodaje element na početak steka (odnosno na početak liste) pa možemo korisiti operaciju dopisivanja elementa na početak liste (#). Ostale operacije se izvršavaju nad dva elementa koja se nalaze na vrhu steka tako što se skidaju ta dva elementa sa steka, izvršava se data operacija, i na kraju se rezultat vraća na stek.

fun *izvrsiOp* :: *Operacija* ⇒ *Stek* ⇒ *Stek* **where**
izvrsiOp (*OpPush x*) *xs* = *x* # *xs* |
izvrsiOp OpPuta (*x* # *y* # *xs*) = (*x* ∗ *y*) # *xs* |
izvrsiOp OpPlus (*x* # *y* # *xs*) = (*x* + *y*) # *xs* |
izvrsiOp OpMinus (*x* # *y* # *xs*) = (*x* − *y*) # *xs*

Prevođenje izraza i generisanje koda za stek mašinu: Sada možemo definisati funkciju koja prevodi dati izraz u listu operacija. Upravo tako ćemo definisati program - lista operacija nad tipom nat. Ovako dobijena lista operacija se čita zdesna na levo.

Prevođenje je manje više trivijalno, konstantni izraz se prevodi u operaciju dodavanja elementa na stek, dok se izraz koji počinje operatorom prevodi u operaciju nad stekom (koja odgovara tom konkretnom operatoru) nakon čega rekurzivno prevodimo njegov prvi pa zatim drugi operand.

type-synonym *Program* = *Operacija list*

primrec *prevedi* :: *Izraz* ⇒ *Program* **where**
prevedi (*Const x*) = [*OpPush x*] |
prevedi (*PlusI i1 i2*) = [*OpPlus*] @ *prevedi i1* @ *prevedi i2* |
prevedi (*MinusI i1 i2*) = [*OpMinus*] @ *prevedi i1* @ *prevedi i2* |
prevedi (*PutaI i1 i2*) = [*OpPuta*] @ *prevedi i1* @ *prevedi i2*

term *prevedi*

Da bismo izgenerisali kod za stek mašinu za dati izraz (*x1* = *PlusI* (*Const 3*) (*Const 5*)), prvo ga prevodimo u listu operacija i dobijamo [*OpPlus*, *OpPush 3*, *OpPush 5*]. Ovako dobijen program čitamo zdesna na levo, što znači da prvo na stek stavljamo broj 5, nakon čega stavljamo broj 3 pa tek onda operator sabiranja.

value *x1*
value *prevedi x1*

U slučaju malo komplikovanijeg izraza *x2*, dobijamo naredni program. Odnosno, prvo na stek stavljamo broj 2, pa broj 5, pa operaciju oduzimanja; nakon toga stavljamo broj 3 pa operaciju množenja.

value *x2* — PutaI (Const 3) (MinusI (Const 5) (Const 2))
value *prevedi x2* — [OpPuta, OpPush 3, OpMinus, OpPush 5, OpPush 2]

Izvršavanje programa: Sada je potrebno definisati rekurzivnu funkciju koja će izvršiti ovako dobijeni program. Da bi se program izvršio potrebno

je da funkcija zna na kom steku ćemo čuvati podatke (na početku je taj stek prazan). Povratna vrednost ovakve funkcije će se naći na vrhu rezultujućeg steka ali to ne možemo tako odmah da zapišemo (zapisaćemo kasnije u okviru leme), pa kao povratni tip očekujemo isto stek.

Ako nemamo više operacija i naš program je prazan, vraćamo tekuće stanje steka. Inače pozivamo funkciju `izvrsiOp` da izvrši operaciju koja se nalazi na početku našeg programa. Tu operaciju ćemo izvršiti nad stekom koji se dobija rekurzivnim pozivom nad repom liste (tj. rekurzivnim pozivom nad ostatkom programa).

primrec *izvrsiProgram* :: *Program* ⇒ *Stek* ⇒ *Stek* **where**
izvrsiProgram [] *s* = *s* |
izvrsiProgram (*op* # *p*) *s* = *izvrsiOp op* (*izvrsiProgram p s*)

Rezultat ovakve funkcije će biti stek koji će imati samo jedan element, i taj element će upravo biti jednak vrednosti prefiksno zadatog izraza.

value *x1*
value *vrednost x1* — rezultat 8
value *prevedi x1* — rezultat [OpPlus, OpPush 3, OpPush 5]
value *izvrsiProgram* (*prevedi x1*) [] — rezultat [8]

Kako je na početku stek prazan, napravićemo školjku oko ovako definisane funkcije i nazvaćemo je `racunar`. Sada ćemo dokazati da je ovakva definicija dobra, odnosno da važi teorema *hd-racunar-vrednost*: *hd* (*racunar i*) = *vrednost i*.

definition *racunar* **where**
racunar i = *izvrsiProgram* (*prevedi i*) []

value *racunar x1* — rezultat [8]

U dokazu leme *hd-racunar-vrednost*, nakon raspisivanja ove definicije dobijamo izraz oblika *hd* (*izvrsiProgram* (*prevedi i*) []) = *vrednost i*. Odnosno imamo poziv funkcije *izvrsiProgram* na početku izvršavanja, kada je stek prazan. Prvo ćemo pokušati sa primenom indukcije (prema definiciji tipa *izraz*) i automatskih alata, i vidimo da se sistem zaustavio kod primene funkcije *izvrsiOp* koja menja stanje steka. Kako je u ovoj lemi početno stanje steka fiksirano na prazan stek, primećujemo da moramo razmotriti šta se dešava kada stek nije prazan, odnosno kako neko početno stanje steka utiče na vrednost izraza koji nam je preostao da izračunamo.

Pošto je izraz koji pokušavamo da dokažemo *hd* (*izvrsiProgram* (*prevedi i*) []) = *vrednost i* u stvari ekvivalentan izrazu *izvrsiProgram* (*prevedi i*) [] = [*vrednost i*], uopštenjem ovog izraza na proizvoljan stek dobijamo lemu *izvrsiProgram-prevedi*: *izvrsiProgram* (*prevedi i*) *s* = *vrednost i* # *s*. Odnosno,

izvršavanje preostalog dela programa ne utiče na prethodno stanje steka (odnosno na podatke koji se već nalaze na steku), već se rezultat dopisuje na početak steka.

U dokazu ove leme ponovo pokušavamo sa indukcijom i automatskim alatima, ali opet dolazimo u situaciju da pozivamo funkciju koja treba da menja stanje steka. Kao što smo ranije videli, ovakva situacija se rešava ključnom rečju *arbitrary*. Sada ponovo pokušavamo sa automatskim dokazivačem i vidimo da je sistem stao kod izraza oblika (*izvrsiProgram* (*prevedi i1* @ *prevedi i2*) *s* pa formulišemo još jednu dodatnu lemu *izvrsiProgram-append*. Leva strana leme se formuliše prema stanju u kom je sistem stao, i sami treba da zaključimo kako izgleda desna strana. Naime pošto smo rekli da se program izvršava tako što se dobijena lista primenjuje zdesna na levo, očekujemo da ova lema tvrdi da se program koji se dobija nadovezivanjem dva programa *p1* i *p2* izvršava tako što se prvo primeni program *p2* (na originalni stek) pa nakon toga program *p1* (na stek dobijen primenom programa *p2*). Ova lema se automatski dokazuje uz pomoć indukcije.

Nakon dodavanja ovih dveju dodatnih lema i njihovog ubacivanja u simplifikator (ključnom rečju *simp*), originalna lema se dokazuje automatski samo uz pomoć simplifikatora.

```
— 5.3
lemma izvrsiProgram-append[simp]:
  shows izvrsiProgram (p1 @ p2) s = izvrsiProgram p1 (izvrsiProgram p2 s)
  apply (induction p1)
   apply auto
  done

— 5.2
lemma izvrsiProgram-prevedi [simp]:
  izvrsiProgram (prevedi i) s = vrednost i # s
— apply (induction i)
— apply auto
  apply (induction i arbitrary: s)
     apply auto
  done

— 5.1 krećemo odavde
theorem hd-racunar-vrednost:
  shows hd (racunar i) = vrednost i
  unfolding racunar-def
— apply (induction i)
— apply auto
```

by *simp*

7.5.2 Prefiksni izraz sa promenljivima

Sada želimo da definišemo prefiksni izraz u kome se mogu javljati i promenljive. Znači pored konstanti tipa možemo imati i promenjive tipa *nat*, ali da bismo naglasili da je to tip koji odgovara promenjivi uvešćemo ga preko sinonima.

Napomena: da ne bismo imali konflikt sa prethodnim definicijama koristićemo nova imena za tipove i operatore.

type-synonym *var-num* = *nat*
datatype *exp* = *Var var-num* | *Con int* | *Add exp exp* | *Mult exp exp* | *Minus exp exp*

Sada je potrebno zadati vrednosti promeljivima koje se javljaju u izrazu. Prvi način je zadavanjem funkcije koja će te promenljive slikati u cele brojeve. Zbog jednostavnosti, pretpostavimo da se u izrazu javljaju samo naredne promenljive: x0 = 1; x1 = 2; x2 = 3. Njihove vrednosti ćemo zadati preko naredne definicije.

definition *val1* :: *var-num* ⇒ *int* **where**
val1 x = (*if x = 0 then 1 else if x = 1 then 2 else 3*)

Sada ćemo definisati funkciju koja izračunava vrednost ovako zadatog izraza. U odnosu na inicijalni primer prefiksnog izraza u kome smo imali samo konstante, ova funkcija mora imati dodatni parametar - funkciju f koja slika promenljive u njihove stvarne vrednost.

primrec *value-exp* :: *exp* ⇒ (*var-num* ⇒ *int*) ⇒ *int* **where**
value-exp (*Var x*) *f* = *f x* |
value-exp (*Con y*) *f* = *y* |
value-exp (*Add exp1 exp2*) *f* = (*value-exp exp1 f*) + (*value-exp exp2 f*) |
value-exp (*Mult exp1 exp2*) *f* = (*value-exp exp1 f*) ∗ (*value-exp exp2 f*) |
value-exp (*Minus exp1 exp2*) *f* = (*value-exp exp1 f*) − (*value-exp exp2 f*)

Ovako definisana funkcija će se pozivati na naredni način. Koristićemo funkciju *val1* koju smo već definisali.

value *value-exp* (*Add* (*Con 2*) (*Var 1*)) *val1* — vrednost 4
definition *x4* :: *exp* **where** *x4* = *Add* (*Con 2*) (*Var 0*)
value *value-exp x4 val1* — vrednost 3

Naredni korak je definisanje dozvoljenih operacija. Kako sada radimo i sa promenljivima, biće potrebno dodati posebnu operaciju koja čita vrednost te promenljive. Stek se definiše na isti način kao ranije, lista (u ovom slučaju) celih brojeva.

datatype *Operation* = *OpAdd* | *OpMult* | *OpMin* | *OpPush int* | *OpRead var-num*

type-synonym *Stack* = *int list*

I prilikom implementacije funkcije koja izvršava jednu operaciju moramo imati dodatni argument, funkciju koja slika promenljive u celobrojne vrednosti (za slučaj kada naiđemo na promenljivu).

fun *executeOp* :: *Operation* ⇒ *Stack* ⇒ (*var-num* ⇒ *int*) ⇒ *Stack* **where**
executeOp OpMult (*x* # *y* # *xs*) *f* = (*x* ∗ *y*) # *xs* |
executeOp OpAdd (*x* # *y* # *xs*) *f* = (*x* + *y*) # *xs* |
executeOp OpMin (*x* # *y* # *xs*) *f* = (*x* − *y*) # *xs* |
executeOp (*OpPush x*) *xs f* = (*x* # *xs*) |
executeOp (*OpRead v*) *xs f* = (*f v* # *xs*)

Program i prevođenje izraza u program se izvršava slično kao ranije.

type-synonym *Prog* = *Operation list*

primrec *translate* :: *exp* ⇒ *Prog* **where**
translate (*Var x*) = [*OpRead x*] |
translate (*Con c*) = [*OpPush c*] |
translate (*Add e1 e2*) = [*OpAdd*] @ *translate e1* @ *translate e2* |
translate (*Minus e1 e2*) = [*OpMin*] @ *translate e1* @ *translate e2* |
translate (*Mult e1 e2*) = [*OpMult*] @ *translate e1* @ *translate e2*

Funkcija koja izvršava program takođe ima kao dodatni argument funkciju koja slika promenljive u celobrojne vrednosti.

primrec *executeProg* :: *Prog* ⇒ *Stack* ⇒ (*var-num* ⇒ *int*) ⇒ *Stack* **where**
executeProg [] *s f* = *s* |
executeProg (*op* # *p*) *s f* = *executeOp op* (*executeProg p s f*) *f*

Poziv ovako definisanih funkcija:

value *x4* — Add (Con 2) (Var 0)
value *value-exp x4 val1* — vrednost 3
value *translate x4* — [OpAdd, Operation.OpPush 2, OpRead 0]
value *executeProg* (*translate x4*) [] *val1* — rezultat [3]

Sada ćemo uvesti narednu definiciju i dokazati (na sličan način kao u prethodnom primeru) da je ta definicija dobro definisana.

definition *computer* **where**
computer i f = *executeProg* (*translate i*) [] *f*

lemma *executeProg-append*[*simp*]:

shows *executeProg (p1 @ p2) s f = executeProg p1 (executeProg p2 s f) f*
apply (*induction p1*)
apply *auto*
done

lemma *executeProg-translate*[*simp*]:
shows *executeProg (translate i) s f = (value-exp i f) # s*
apply (*induction i arbitrary*: *s*)
apply *auto*
done

theorem *hd*(*computer i f*) = *value-exp i f*
unfolding *computer-def*
by *simp*

7.5.3 Interpretator koji izračunava vrednost izraza

Prethodno rešenje sa korišćenjem funkcije se može uopštiti. Sada ćemo prikazati kako možemo vrednosti promenljivih da čuvamo u mapi koja će biti korišćena da imena promenljivih preslika u njihove stvarne vrednosti.

type-synonym *var-num′* = *nat*
datatype *exp′* = *Var var-num′* | *Const int* | *Add exp′ exp′* | *Mult exp′ exp′* | *Minus exp′ exp′*

— Čitanje iz memorije ćemo implementirati preko tabele koja će biti predstavljena uz pomoć mapa. Uključujemo teoriju *HOL−Library.Mapping*.
https://isabelle.in.tum.de/library/HOL/HOL-Library/Mapping.html

Mapa je predstavljena kao skup uređenih parova, što zapisujemo na sledeći način:

type-synonym *Memory* = (*nat*, *int*) *mapping*

Za čitanje iz mape koristićemo funkciju *lookup-default* koja je implementirana tako da vraća svoj prvi argument (podrazumevanu, default vrednost) ako u mapi ne postoji uređeni par koji odgovara vrednosti koja se traži.

definition *read-memory* **where**
read-memory m x = Mapping.lookup-default 0 m x

Funkcija koja izračunava vrednost izraza tako što čita vrednosti promenljivih iz memorije se može implementirati na sledeći način:

primrec *value-exp-map*:: *exp′* ⇒ *Memory* ⇒ *int* **where**

value-exp-map (Var x) m = read-memory m x |
value-exp-map (Const y) m = y |
value-exp-map (Add exp1 exp2) m = (value-exp-map exp1 m) + (value-exp-map exp2 m) |
value-exp-map (Mult exp1 exp2) m = (value-exp-map exp1 m) ∗ (value-exp-map exp2 m) |
value-exp-map (Minus exp1 exp2) m = (value-exp-map exp1 m) − (value-exp-map exp2 m)

Za popunjavanje memorije koristimo funkciju

```
bulkload :: "'a list \<Rightarrow> (nat, 'a) mapping"
```

koja datu listu preslikava u stanje u memoriji. Ovaj poziv odgovara vrednostima promenljivih koje smo koristili u prethodnom primeru.

definition *val-map1* :: *Memory* **where**
val-map1 = Mapping.bulkload [1,2,3]

Uključujemo RBT stabla da bi moglo da izvrši računanje, teoriju *RBT-Mapping*.

value *value-exp-map (Mult (Var 2) (Add (Const 2) (Const 5))) val-map1*

definition *x5* :: *exp′* **where** *x5 = Add (Var 0) (Const 3)*
value *value-exp-map x5 val-map1*

definition *x6* :: *exp′* **where** *x6 = Mult (Var 0) (Add (Const 2) (Const 5))*
value *value-exp-map x6 val-map1*

Uvodimo novu operaciju čitanja iz memorije:

datatype *Operation′ = OpAdd | OpMult | OpMin | OpPush int | OpRead var-num*

type-synonym *Stack′ = int list*

fun *executeOp′* :: *Operation′* ⇒ *Stack′* ⇒ *Memory* ⇒ *Stack′* **where**
executeOp′ OpMult (x # y # xs) m = (x ∗ y) # xs |
executeOp′ OpAdd (x # y # xs) m = (x + y) # xs |
executeOp′ OpMin (x # y # xs) m = (x − y) # xs |
executeOp′ (OpPush x) xs m = (x # xs) |
executeOp′ (OpRead x) xs m = ((read-memory m x) # xs)

Uvodimo program, prevođenje izraza u program i njegovo izvršavanje:

type-synonym *Prog′ = Operation′ list*

primrec *translate′* :: *exp′* ⇒ *Prog′* **where**
translate′ (Var x) = [OpRead x] |

```
translate' (Const x) = [OpPush x] |
translate' (Add i1 i2) = [OpAdd] @ translate' i1 @ translate' i2 |
translate' (Minus i1 i2) = [OpMin] @ translate' i1 @ translate' i2 |
translate' (Mult i1 i2) = [OpMult] @ translate' i1 @ translate' i2

primrec executeProg' :: Prog' ⇒ Stack' ⇒ Memory ⇒ Stack' where
executeProg' [] s m = s |
executeProg' (op # p) s m = executeOp' op (executeProg' p s m) m

value x5
value value-exp-map x5 val-map1
value translate' x5
value executeProg' (translate' x5) [] val-map1
```

Na sličan način kao u prethodna dva primera uvodimo narednu definiciju i dokazujemo da je dobro definisana:

```
definition computer-map where
computer-map i m = executeProg' (translate' i) [] m

lemma executeProg'-append[simp]:
  shows executeProg' (p1 @ p2) s m = executeProg' p1 (executeProg' p2 s m) m
  apply (induction p1)
   apply auto
done

lemma executeProg'-translate[simp]:
  shows executeProg' (translate' i) s m = (value-exp-map i m) # s
  apply (induction i arbitrary: s)
     apply auto
done

theorem hd(computer-map i m) = value-exp-map i m
  unfolding computer-map-def
by simp

end
```

7.6 Aksiomatsko zasnivanje teorija, korišćenje lokala

theory *Cas14-vezbe*
imports *Complex-Main HOL−Library.Code-Target-Nat HOL−Library.Multiset*

begin

Aksiomatsko rezonovanje, koje smo navikli da viđamo u matematici, ima svoje primene i u verifikaciji softvera. Jedan deo formalne semantike programskih jezika se obrađuje aksiomatskim pristupom. Pod ovim se podrazumeva definisanje svojstava funkcije, odnosno *dovoljnih uslova* koje funkcija treba da ispunjava i nezavisno dokazivanje teorema koje opisuju ta svojstva uz pomoć interaktivnih i automatskih dokazivača teorema.

U ovom poglavlju ćemo prikazati primenu aksiomatskog metoda u razvoju formalno verifikovanog softvera. Korektnost sofvera je lakše dokazati ako se razvija korak po korak, umesto odjednom. Prvo kreiramo specifikaciju programa i program koji neke detalje ostavlja nerazrešenim i ostavljamo njihovu implementaciju za kasnije dok ne dođemo do konačne izvršive verzije.

Ovaj pristup se naziva razvoj softvera profinjavanjem *eng: refinement* (izučava se od 70-tih godina), i nije ograničen na formalno verifikovanje softvera. Krećemo od glavne funkcije na veoma apstraktnom nivou i specifikujemo sve više i više detalja softvera dok ne dođemo do konačne verzije.

Pogledaćemo kako izgleda ovaj pristup na primeru implementacije funkcije sortiranja objedinjavanjem koji je obrađen u materijalu iz (pretprošlog) poglavlja 7.4.

7.6.1 Sortiranje objedinjavanjem

Ponovimo prvo definiciju funkciju objedinjavanja dve sortirane liste, i svojstva koja ta funkcija mora da zadovoljava:

fun *spoji* :: *nat list* ⇒ *nat list* ⇒ *nat list* **where**
spoji [] *ys* = *ys* |
spoji xs [] = *xs* |
spoji (*x* # *xs*) (*y* # *ys*) =
  (*if x* ≤ *y then*
    x # *spoji xs* (*y* # *ys*)
   else
    y # *spoji* (*x* # *xs*) *ys*)

value *spoji* [*2*, *5*, *8*, *13*] [*1*, *3*, *9*, *34*, *40*]

Ponavljamo dokaze da su multiskup i skup očuvani, kao i da je rezultujuća lista sortirana. Pošto smo prve dve leme dodali u simplifikator, treća lema prolazi automatski.

lemma *mset-spoji*[*simp*]:
shows *mset* (*spoji xs ys*) = *mset xs* + *mset ys*
by (*induction xs ys rule*: *spoji.induct*, *auto*)

lemma *set-spoji*[*simp*]:
shows *set* (*spoji xs ys*) = *set xs* ∪ *set ys*
by (*induction xs ys rule*: *spoji.induct*, *auto*)

lemma *sorted-spoji* [*simp*]:
assumes *sorted xs sorted ys*
shows *sorted* (*spoji xs ys*)
using *assms*
by (*induction xs ys rule*: *spoji.induct*) *auto*

Prilikom implementacije sortiranja objedinjavanjem koristiće se ova funkcija. Međutim, u funkciji sortiranja prvi korak je deljenje liste na dva dela, nakon čega sledi rekurzivno sortiranje ta dva dela i nakon toga sledi objedinjavanje funkcijom koja spaja dve sortirane liste.

Zbog efikasnosti koda, ta dva dela liste bi trebala da budu jednake dužine. Ali, primetimo da je ova pretpostavka o dužinama ta dva dela liste vezana samo za efikasnost sortiranja ali ne i za korektnost funkcije sortiranja.

Sada je na redu korak u kome treba da implementiramo neki algoritam deljenja originalne liste na dve liste. Postoji više načina kako to možemo da uradimo: npr. prva polovina liste i druga polovina liste, ili elementi na neparnim i elementi na parnim pozicijama i slično. Međutim, ispostavlja se da pitanje deljenja liste na dve liste, odnosno sam izbor algoritma za deljenje na dve liste možemo ostaviti za kasnije. Za sada je potrebno da primetimo da je neophodno da algoritam deljenja liste ima naredne dve osobine:

- korektnost - da je podela na dve liste urađena tako da su tačno svi elementi liste raspoređeni u njih i,
- zaustavljanje - da su obe nove liste kraće od polazne liste.

Ove osobine deljenja liste ćemo nadalje nazivati aksiome.

Pre nego što preciziramo algoritam deljenja, možemo dokazati malo jači oblik funkcije sortiranja objedinjavanjem. Dokazaćemo korektnost funkcije objedinjavanja u odnosu na apstraktnu funkciju podele liste na dva dela. Funkciju podele na dve liste ćemo zadati apstraktno pomoću njene specifikacije.

Koristimo nov koncept *lokala* (engl. locale) - to je skup konstanti, odnosno skup pojedinačnih elemenata nekog tipa ili funkcije. Prilikom zadavanja lokala, prvo se fiksiraju određeni elementi nakon čega se navode njihove osobine kroz navođenje nekih aksioma. U narednom lokalu ćemo fiksirati funkciju *split* (ključnom rečju `fixes`) koja od date liste pravi uređeni par listi. Nakon ovoga zadajemo aksiome (ključnom rečju `assumes`) koje moraju da važe da bi algoritam sortiranja objedinjavanjem koji koristi ovu funkciju *split* mogao da se isprogramira. To su aksiome koje obezbeđuju korektnost algoritma i njegovo zaustavljanje.

Umesto običnog univerzalnog kvantifikatora ($\forall$), implikacije ($\longrightarrow$) i konjunkcije ($\wedge$) koristimo veznike meta-logike i Isabelle sintaksu zapisa ($\bigwedge$), ($\Longrightarrow$) i ($[\![,]\!]$), zbog kasnijeg lakšeg instanciranja ovih aksioma.

Pošto se lista deli na polovine samo ako ima više od jednog elementa, dovoljno je zahtevati da ove aksiome važe samo za takve liste, pa uslov da je dužina liste veća od 1 stavljamo u pretpostavke obe aksiome.

Nakon navođenja aksioma, između ključnih reči `begin` i `end` navodi se blok koji sadrži lokalne definicije i teoreme i u kome smemo da koristimo aksiome koje su uvedene. Na ovom mestu se definiše algoritam sortiranja objedinjavanjem. Kako nam u algoritmu treba poređenje elemenata, da bismo pojednostavili primer zapisaćemo definiciju za liste prirodnih brojeva. Ova definicija i dokaz korektnosti je isti kao u prošlom poglavlju pa ga nećemo ponovo objašnjavati.

locale *MergeSort* =
— prvo se navode konstante
fixes *split* :: *nat list* $\Rightarrow$ *nat list* $\times$ *nat list*
— pa onda aksiome
assumes *split-length*:
$\bigwedge$ *xs ys zs*. $[\![$*length xs* > *1*; (*ys*, *zs*) = *split xs*$]\!] \Longrightarrow$
length ys < *length xs* $\wedge$ *length zs* < *length xs*
assumes *split-mset*:
$\bigwedge$ *xs ys zs*. $[\![$*length xs* > *1*; (*ys*, *zs*) = *split xs*$]\!] \Longrightarrow$
mset xs = *mset ys* + *mset zs*

— Između begin i end uvode se lokalne definicije i dokazuju se lokalna tvrđenja (u njima mogu da se koriste konstante i aksiome lokala).

begin

function (*sequential*) *mergesort* :: *nat list* ⇒ *nat list* **where**
mergesort [] = [] |
mergesort [*x*] = [*x*] |
mergesort xs =
  (*let* (*ys*, *zs*) = *split xs*
    in spoji (*mergesort ys*) (*mergesort zs*))
 by *pat-completeness auto*
termination
 apply (*relation measure* (λ *xs*. *length xs*))
  apply *simp*
 apply (*metis One-nat-def in-measure length-Cons lessI linorder-not-less list.size*(*4*) *split-length trans-le-add2*)
 apply (*metis One-nat-def impossible-Cons in-measure linorder-not-less list.size*(*4*) *split-length trans-le-add2*)
 done

— Unutar lokala ne možemo pozvati funkciju *mergesort* jer funkcija *split* nije definisana. Ali možemo pokazati da važe još neke osobine te funkcije: da se mset ne menja i da je rezultujuća lista sortirana:

lemma *mset-mergesort-auto*:
 mset (*mergesort xs*) = *mset xs*
 apply (*induction xs rule*: *mergesort.induct*)
 apply *simp*
 apply *simp*
  apply (*auto split*: *prod.split*)
 — sledgehammer
 by (*metis* (*mono-tags*, *hide-lams*) *One-nat-def Suc-eq-plus1 length-Cons less-Suc-eq-0-disj mset.simps*(*2*) *mset-append split-mset zero-less-Suc*)

lemma *mset-mergesort*:
 mset (*mergesort xs*) = *mset xs*
proof (*induction xs rule*: *mergesort.induct*)
 case *1*
 then show *?case*
  by *simp*
next
 case (*2 x*)
 then show *?case*
  by *simp*

```
next
  case (3 x0 x1 xs)
  then show ?case
    using split-mset[of x0 # x1 # xs]
    by (simp split: prod.split)
qed

lemma sorted-mergesort:
  sorted (mergesort xs)
  by (induction xs rule: mergesort.induct) (auto split: prod.split)
end
```

Dokazali smo da je funkcija mergesort korektna ako funkcija `split` zadovoljava navedene aksiome koje su ekspliticno navedene u prethodnom lokalu. Primetimo da je ovo urađeno bez implementacije funkcije `split`. U ovom trenutku imamo slobodu da razrešimo detalje te implementacije.

Sada ćemo prikazati dve različite implementacije funkcije `split`.

Razdvajanje uzimanjem prve i druge polovine niza

Naredna funkcija izdvaja, na osnovu dužine liste, prvu polovinu odnosno drugu polovinu liste.

```
definition split-half :: nat list ⇒ nat list × nat list where
  split-half xs = (let n = length xs div 2 in (take n xs, drop n xs))

value split-half [3::nat, 4, 8, 1, 2, 0, 5]
```

Sada nam treba način da povežemo ovu konretnu implementaciju sa opštom implementacijom algoritma za sortiranje koji smo naveli uz pomoć lokala.

Koristimo interpretaciju lokala i ključnu reč `global_interpretation` nakon čega navodimo novo ime lokala, pa ime osnovnog lokala iz koga izvodimo novi lokal, ključnu reč `where` i ime apstraktne funkcije (konstante lokala) koju sada zamenjujemo konkretnom funkcijom koju smo sada implementirali. Za tu konkretnu funkciju treba da dokažemo da važe aksiome koje su navedene u lokalu, odnosno treba da dokažemo da ovakva funkcija podele zadovoljava potrebna svojstva. Ovaj dokaz se navodi između `proof` i `qed` bloka, pri čemu su dokazi različitih aksioma razdvojeni ključnom rečju `next`.

Napomenimo da je red koji počinje sa ključnom rečju `defines` neophodan da bi funkcija *mergesort-half* mogla da se izvršava. Nakon ključne reči se navodi ime funkcije (koje ćemo koristiti u nastavku), ime funkcije iz lokala i konkretnu funkciju za deljenje liste koju ćemo koristiti u ovoj interpretaciji.

```
global-interpretation MergeSort-half: MergeSort where
  split = split-half
  — Naredni red je potrebno navesti da bi funkcija mergesort mogla da se izvršava
  defines mergesort-half = MergeSort.mergesort split-half
proof
  fix xs ys zs :: nat list
  assume length xs > 1 (ys, zs) = split-half xs
  then show length ys < length xs ∧ length zs < length xs
    unfolding split-half-def Let-def
    by auto
next
  fix xs ys zs :: nat list
  assume length xs > 1 (ys, zs) = split-half xs
  then show mset xs = mset ys + mset zs
    unfolding split-half-def Let-def
    — sledgehammer
    by (simp add: union-code)
qed
```

Sada možemo pozvati ovako intepretiranu funkciju:

```
value mergesort-half [3, 8, 4, 2, 1]
```

Ali imamo i dve teoreme dobijene u ovom lokalu. Jedna teorema koja tvrdi da zaista ova kombinacija sortira listu i druga koja tvrdi da je multiskup elemenata nepromenjen.

```
thm MergeSort-half.mset-mergesort
thm MergeSort-half.sorted-mergesort
```

Ovako interpretirana funkcija može da se eksportuje u neki drugi programski jezik:

```
export-code mergesort-half in Haskell
```

Ovde ponovo možemo da primetimo problem sa brzinom izvršavanja ovakvog koda zbog načina na koji su implementirani prirodni brojevi i gubi se puno vremena na poređenje prirodnih brojeva (koji su predstavljeni preko Suc). Tako da je u situacijama kada želimo da eksportujemo kod potrebno uključiti na početku fajla teoriju `Code_Target_Nat`.

```
value length (mergesort-half [0..<4000])
```

Razdvajanje naizmeničnim uzimanjem elemenata

Pokazaćemo kako izgleda interpretacija ovog istog lokala sa drugom funkcijom za deljenje na dve liste, kada uzimamo elemente naizmenično. Funkciju

ćemo nazvati `split_zigzag`, definisaćemo je rekurzijom i dokazi će koristiti indukciju po pravilu `split_zigzag.induct`.

```
fun split-zigzag :: nat list ⇒ nat list × nat list where
split-zigzag [] = ([], []) |
split-zigzag [x] = ([x], []) |
split-zigzag (x0 # x1 # xs) =
      (let (ys, zs) = split-zigzag xs
        in (x0 # ys, x1 # zs))

value split-zigzag [3::nat, 2, 8, 4, 1, 5, 6]

global-interpretation MergeSort-zigzag: MergeSort
  where split = split-zigzag
  — Naredni red je potrebno navesti da bi funkcija mergesort-zigzag mogla da se izvršava
  defines mergesort-zigzag = MergeSort.mergesort split-zigzag
proof
  fix xs ys zs :: nat list
  assume length xs > 1 (ys, zs) = split-zigzag xs
  then show length ys < length xs ∧ length zs < length xs
  proof (induction xs arbitrary: ys zs rule: split-zigzag.induct)
    case 1
    then show ?case
      by simp
  next
    case (2 x)
    then show ?case
      by simp
  next
    case (3 x0 x1 xs)
    show ?case
    proof (cases xs = [])
      case True
      thus ?thesis
        using 3(3)
        by simp
    next
      case False
      show ?thesis
      proof (cases length xs = 1)
        case True
        then obtain x where xs = [x]
          by (metis One-nat-def length-0-conv length-Suc-conv)
```

```
      thus ?thesis
        using 3(3)
        by simp
    next
      case False
      then have length xs > 1
        using ‹xs ≠ []›
        using not-less-iff-gr-or-eq by auto
      then obtain ys' zs' where ∗: (ys', zs') = split-zigzag xs
        by (metis surj-pair)
      then have ∗∗: ys = x0 # ys' zs = x1 # zs'
        using 3(3)
        by (metis Pair-inject prod.simps(2) split-zigzag.simps(3))+
      then have length ys' < length xs ∧ length zs' < length xs
        using ∗ 3(1) ‹length xs > 1›
        by simp
      then show ?thesis
        using ∗∗
        by simp
    qed
  qed
 qed
next
 fix xs ys zs :: nat list
 assume (ys, zs) = split-zigzag xs
 then show mset xs = mset ys + mset zs
   by (induction xs arbitrary: ys zs rule: split-zigzag.induct)
     (auto split: prod.split-asm)
qed
```

Ovako interpretirana funkcija se sada može pozvati:

```
value mergesort-zigzag [3, 8, 1, 4, 2, 5, 6]
```

I opet imamo dve teoreme koje tvrde da ova funkcija zaista sortira listu i da je multiskup elemenata nepromenjen:

```
thm MergeSort-zigzag.mset-mergesort
thm MergeSort-zigzag.sorted-mergesort
```

7.6.2 Apstrakcija topološkog prostora

Sličan pristup, uvođenje apstraktnog modela, dokazivanje određenih svojstava i naknadno instanciranje se primenjuje i u matematici. Na taj način možemo izbeći ponavljanje dokaza određenih osobina sličnih struktura. De-

taljnije, ovim postižemo da uvođenjem odgovarajućih apstrakcija i dokazivanjem teorema u apstraktnom obliku, dobijamo opciju da upotrebimo konkretne instance dokazanih teorema na strukturama za koje je dokazano da zadovoljavaju osobine odgovarajuće apstrakcije.

Na primer, umesto da se zajedničke teoreme dokažu prvo za polje realnih, a zatim za polje kompleksnih brojeva, teoreme se mogu dokazati na apstraktnom nivou, za sva polja. Nakon toga, pošto pokažemo da, na primer, realni brojevi čine polje u odnosu na odgovarajuće operacije, sve teoreme dokazane za polja važiće i u odgovarajućoj strukturi realnih brojeva.

Ilustrujmo ovaj postupak uvođenjem apstrakcije topološkog prostora.[2][3]

Topološki prostor (na nekom skupu X) je par (X,τ), gde X predstavlja dati skup, a τ predstavlja familiju podskupova skupa X. Podskupove koji pripadaju familiji τ nazvaćemo otvorenim podskupovima. Ova familija treba da zadovoljava sledeće uslove:

- Prazan skup i ceo skup X moraju biti otvoreni skupovi — aksiome *empty* i *univ*
- Unija proizvoljnog broja otvorenih skupova je otvoren skup — aksioma *union*
- Presek dva otvorena skupa je otvoren skup — aksioma *inter*

Pored ove četiri aksiome potrebno je eksplicitno navesti da elementi skupa τ moraju biti podskupovi skupa X (obrnuto neće važiti), i to tvrđenje ćemo predstaviti u narednom lokalu aksiomom sa imenom *subsets*. Napomenimo još sintaksu koju ćemo koristiti. Skup S je element skupa τ pišemo uobičajeno $S \in \tau$. Zanimljivo je to da proizvoljan broj otvorenih skupova u stvari predstavlja podskup skupa τ, pa ćemo rečenicu tipa skup τ' sadrži proizvoljan broj otvorenih skupova jednostavno zapisivati sa $\tau' \subseteq \tau$. A uniju nad tim skupom, odnosno uniju proizvoljnog broja otvorenih skupova ćemo zapisati sa $\bigcup \tau'$.

Formalizacija ovoj pojma vrši se kroz koncept lokala. Lokal uvodi određeni broj konstanti/funkcija (koje se kasnije konkretizuju) i aksioma koje te konstante/funkcije moraju da zadovolje. Svaki topološki prostor fiksira konstantu X (skup nosilac) i konstantu τ (skup svih otvorenih podskupova).

[2]Jedna odlična, veoma lako čitljiva knjiga o topologiji: http://www.topologywithouttears.net/topbook.pdf.

[3]Malo kraći materijal o topološkim prostorima može se naći na: https://www.dmi.uns.ac.rs/site/dmi/download/master/primenjena_matematika/MilijanaMilovanovic.pdf

Ovako definisana familija τ se naziva i *topologija* nad skupom X.

Može se pokazati da je familija τ zatvorena u odnosu na konačne preseke. Ovo tvrđenje ćemo nezavisno formulisati i dokazati kao lemu u narednom lokalu (lema *inter-finite*). Da bi smo proverili da li je skup konačan koristićemo funkciju *finite* :: *'a set* ⇒ *bool* koja vraća vrednost *True* ako je njen argument konačan. Dokaz te leme će se izvoditi indukcijom nad konačnim skupom τ'. Pravilo indukcije na koje se pozivamo se u ovom slučaju naziva *finite.induct*: ⟦*finite x*; *P* {}; ⋀*A a*. ⟦*finite A*; *P A*⟧ ⟹ *P* (*insert a A*)⟧ ⟹ *P x* i ovako se zapisuje u Isabelle/HOL. U dokazu ćemo razlikovati slučaj praznog skupa i slučaj nepraznog skupa (koji se dobija ubacivanjem elementa u skup funkcijom *insert* :: *'a* ⇒ *'a set* ⇒ *'a set*).

Pored pojma otvorenog skupa, koji smo definisali kao elemente familije τ, uvodimo i pojam *zatvorenog skupa* - skup je zatvoren ako i samo ako je njegov komplement u odnosu na skup X otvoren skup. Ova dva pojma ćemo uvesti pojedinačno u narednom lokalu.

Primetimo da se na sličan način može definisati i topologija uz pomoć zatvorenih skupova. To možemo i dokazati na osnovu već navedenih aksioma za otvorene skupove i de Morganovih zakona. Tako da ćemo formulisati (i dokazati) dualne teoreme za zatvorene skupove: prazan skup i ceo prostor su zatvoreni skupovi (leme *closed-empty* i *closed-univ*), presek dva zatvorena skupa je zatvoren skup (lema *closed-inter*), presek konačno mnogo zatvorenih skupova je zatvoren skup (lema *closed-inter-finite*) i unija konačno mnogo zatvorenih skupova je zatvoren skup (lema *closed-union-finite*).

```
locale topological-space =
  — konstante
  fixes X :: 'a set  — skup nosilac
  fixes τ :: 'a set set — kolekcija otvorenih skupova

  — aksiome
  assumes subsets: ⋀ S. S ∈ τ ⟹ S ⊆ X
  assumes univ: X ∈ τ
  assumes empty: {} ∈ τ
  assumes inter: ⋀ S1 S2. ⟦S1 ∈ τ; S2 ∈ τ⟧ ⟹ S1 ∩ S2 ∈ τ
  assumes union: ⋀ τ'. ⟦τ' ≠ {}; τ' ⊆ τ⟧ ⟹ ⋃ τ' ∈ τ
begin

— Skup je otvoren ako i samo ako pripada τ.
abbreviation open-set :: 'a set ⇒ bool where
  open-set S ≡ S ∈ τ

— U narednom dokazu ćemo koristiti pravilo indukcije nad konačnim skupom.
```

term *finite*
thm *finite.induct*

— Dokazujemo da je presek konačno mnogo otvorenih skupova otvoren skup:
lemma *inter-finite*:
  assumes *finite* τ' $\tau' \neq \{\}$ $\forall\ S \in \tau'$. *open-set* S
  shows *open-set* $(\bigcap\ \tau')$
  using *assms*
proof (*induction* τ' *rule*: *finite.induct*)
  case *emptyI*
  then show *?case*
    by *simp*
next
  case (*insertI* A a)
  thm *insertI* — obratite pažnju na to kako izgleda korak indukcije u ovakvom dokazu
  term *insert* — funkcija koja ubacuje element u skup
  show *?case*
  proof (*cases* $A = \{\}$)
    case *True*
    then show *?thesis*
      using *insertI*
      by *simp*
  next
    case *False*
    then show *?thesis*
      using *insertI inter* — pozivamo se na induktivnu pretpostavku i aksiomu inter
      by *simp*
  qed
qed

— Skup je zatvoren ako i samo ako mu je komplement (u odnosu na X) otvoren:
definition *closed-set* :: *'a set* ⇒ *bool* **where**
  closed-set $S \longleftrightarrow$ *open-set* $(X - S)$

— Zatvoreni skupovi takođe definišu topologiju nad skupom X:
lemma *closed-empty*:
  closed-set $\{\}$
  unfolding *closed-set-def*
  using *univ*
  by *simp*

lemma *closed-univ*:
closed-set X
unfolding *closed-set-def*
using *empty*
by *simp*

lemma *closed-inter*:
assumes *closed-set S1 closed-set S2*
shows *closed-set* (*S1* ∩ *S2*)
using *assms union*[*of* {*X* − *S1*, *X* − *S2*}]
unfolding *closed-set-def*
by (*simp add*: *Diff-Int*)

lemma *closed-inter-finite*:
assumes *finite* τ' $\tau' \neq \{\}$ ∀ *S* ∈ τ'. *closed-set S*
shows *closed-set* ($\bigcap$ τ')
proof−
have *X* − ($\bigcap$ τ') = $\bigcup$ {*X* − *S* | *S* . *S* ∈ τ'}
using *assms*
by *auto*
thus *?thesis*
using *assms*
unfolding *closed-set-def*
using *union*[*of* {*X* − *S* |*S*. *S* ∈ τ'}]
by *auto*
qed

lemma *closed-union-finite*:
assumes *finite* τ' $\tau' \neq \{\}$ ∀ *S* ∈ τ'. *closed-set S*
shows *closed-set* ($\bigcup$ τ')
proof−
have *X* − $\bigcup$ τ' = $\bigcap$ {*X* − *S* | *S* . *S* ∈ τ'}
using ‹$\tau' \neq \{\}$›
by *auto*
thus *?thesis*
using *assms*
using *inter-finite*[*of* {*X* − *S* | *S*. *S* ∈ τ'}]
unfolding *closed-set-def*
by *auto*
qed

end

Ograničen skup prirodnih brojeva

U nastavku je dat jedan (prilično veštački) primer topološkog prostora:

definition *X-ex1* :: *nat set* **where**
X-ex1 = {*0*::*nat*, *1*, *2*, *3*, *4*, *5*}

definition *τ-ex1* :: *nat set set* **where**
τ-ex1 = {{*0*::*nat*, *1*, *2*, *3*, *4*, *5*}, {}, {*0*}, {*2*, *3*}, {*0*, *2*, *3*}, {*1*, *2*, *3*, *4*, *5*}}

Komandom `global_interpretation` dokazujemo da neke konkretne konstante interpretiraju ranije uvedenu apstrakciju. Odnosno, treba da pokažemo (da ih formalno dokažemo) da važe aksiome koje smo naveli u lokalu.

declare [[*quick-and-dirty*=*true*]]

global-interpretation *topological-space* **where**
X = *X-ex1* **and**
τ = *τ-ex1*
proof
fix *S*
assume *S* ∈ *τ-ex1*
thus *S* ⊆ *X-ex1*
unfolding *τ-ex1-def X-ex1-def*
by *auto*
next
show *X-ex1* ∈ *τ-ex1* {} ∈ *τ-ex1*
unfolding *X-ex1-def τ-ex1-def*
by *auto*
next
fix *S1 S2*
assume *S1* ∈ *τ-ex1 S2* ∈ *τ-ex1*
thus *S1* ∩ *S2* ∈ *τ-ex1*
unfolding *τ-ex1-def*
— ovi dokazi su krajnje neinteresantni, a ne mogu se dokazati automatski
sorry
next
fix *τ'*
assume *τ'* ≠ {} *τ'* ⊆ *τ-ex1*
thus ⋃ *τ'* ∈ *τ-ex1*
unfolding *τ-ex1-def*
— ovi dokazi su krajnje neinteresantni, a ne mogu se dokazati automatski
sorry
qed

Skup realnih brojeva

Mnogo interesantniji primer topološkog prostora je skup realnih brojeva sa tzv. euklidskom topologijom.

Otvoreni interval (a, b) se zadaje na sledeći način:

```
definition open-interval :: real ⇒ real ⇒ real set where
  open-interval a b = {x. a < x ∧ x < b}

lemma open-interval-empty-iff:
  open-interval a b = {} ⟷ a ≥ b
  unfolding open-interval-def
  by (metis (no-types, lifting) dense-le dual-order.strict-implies-order
  empty-Collect-eq leD le-less-linear order.strict-trans1)
```

Nakon ovoga ćemo definisati otvoren skup - skup je otvoren ako i samo ako se svaka njegova tačka može okružiti nekim otvorenim intervalom koji ceo pripada tom skupu.

```
definition is-real-open-set where
  is-real-open-set S ⟷
      (∀ x ∈ S. ∃ a b. x ∈ open-interval a b ∧ open-interval a b ⊆ S)
```

Dokazujemo da skup realnih brojeva sa ovako uvedenom definicijom otvorenog skupa zaista predstavlja jedan topološki prostor.

U narednoj interpretaciji se koristi UNIV - skup svih elemenata nekog tipa.

```
global-interpretation Euclidean-topology: topological-space where
 X = UNIV :: real set  and
 τ = {S. is-real-open-set S}
proof
  fix S
  assume S ∈ {S. is-real-open-set S}
  then show S ⊆ UNIV
    by simp
next
  have ∀ x. ∃ a b. x ∈ open-interval a b
  proof−
    have ∀ x. x ∈ open-interval (x − 1) (x + 1)
      unfolding open-interval-def
      by simp
    then show ?thesis
      by auto
  qed
  then show UNIV ∈ {S. is-real-open-set S}
```

```
    unfolding is-real-open-set-def
    by auto
next
  show {} ∈ {S. is-real-open-set S}
    unfolding is-real-open-set-def
    by simp
next
  fix τ'
  assume ∗: τ' ⊆ {S. is-real-open-set S}
  show ⋃ τ' ∈ {S. is-real-open-set S}
  proof
    show is-real-open-set (⋃ τ')
      unfolding is-real-open-set-def
    proof
      fix x
      assume x ∈ ⋃ τ'
      then obtain S where S ∈ τ' x ∈ S
        by auto
      hence is-real-open-set S
        using ∗
        by auto
      then obtain a b where x ∈ open-interval a b open-interval a b ⊆ S
        using ⟨x ∈ S⟩
        unfolding is-real-open-set-def
        by auto
      thus ∃ a b. x ∈ open-interval a b ∧ open-interval a b ⊆ ⋃ τ'
        using ⟨S ∈ τ'⟩
        by auto
    qed
  qed
next
  fix S1 S2
  assume ∗: S1 ∈ {S. is-real-open-set S} S2 ∈ {S. is-real-open-set S}
  show S1 ∩ S2 ∈ {S. is-real-open-set S}
  proof
    show is-real-open-set (S1 ∩ S2)
      unfolding is-real-open-set-def
    proof
      fix x
      assume x ∈ S1 ∩ S2
      then obtain a1 b1 a2 b2 where
      ∗∗: x ∈ open-interval a1 b1 open-interval a1 b1 ⊆ S1
          x ∈ open-interval a2 b2 open-interval a2 b2 ⊆ S2
```

```
      using ∗
      unfolding is-real-open-set-def
      by blast
    let ?a = max a1 a2
    let ?b = min b1 b2
    have x ∈ open-interval ?a ?b
      using ∗∗
      unfolding open-interval-def
      by auto
    moreover
    have open-interval ?a ?b ⊆ S1 ∩ S2
      using ∗∗
      unfolding open-interval-def
      by auto
    ultimately
    show ∃ a b. x ∈ open-interval a b ∧ open-interval a b ⊆ S1 ∩ S2
      by auto
  qed
 qed
qed
```

Kada smo dokazali da realni brojevi predstavljaju topološki prostor, na raspolaganju imamo sve definicije i teoreme dokazane o topološkim prostorima.

```
thm Euclidean-topology.closed-set-def
thm Euclidean-topology.closed-inter
thm Euclidean-topology.closed-union-finite

lemma Euclidean-topology.open-set A ⟷ is-real-open-set A
 by simp


end
```

Appendix A

Priprema dokumenata

theory *Cas15-vezbe*
imports *HOL−Library.LaTeXsugar HOL−Library.OptionalSugar*

begin

U ovom poglavlju biće dat kratak pregled mogućnosti koje Isabelle/HOL nudi vezano sa formatiranje teksta, generisanje html i Latex dokumenata.

Prilikom kucanja teksta u Isabelle/HOL može se namestiti maksimalna dužina linije naredbom (∗:*maxLineLen*=*80*:∗). Potrebno je namestiti i opciju Soft wrap, pod menijem Utilities/Global Options/Word wrap. Ovakvo ograničenje na dužinu linije će biti vidljivo samo u programu Isabelle/HOL. Prilikom transformisanja dokumenta u druge formate ovo ograničenje neće biti preneseno. http://www.jedit.org/users-guide/word-wrap.html

Pored samih teorema i njihovih dokaza propratni tekst može biti unet uz pomoć sledećih opcija:

- običnih komentara između (∗ i ∗)
- marginalnih komentara koji se kucaju nakon naredbe `\<comment>` koja se prikazuje kao —
- dužeg teksta koji se navodi u okviru naredbe `text \<open> \<close>` a koja se prikazuje kao *text* ‹›

A.1 Generisanje dokumenata

A.1.1 Generisanje html dokumenta

Za kreiranje html dokumenta od postojećeg thy fajla, dovoljno je izabrati opciju Plugins/Code2HTML/HTMLize current buffer. Nakon toga dobiće se izgenerisani html kod koji je potrebno sačuvati pod željenim imenom.

A.1.2 Generisanje pdf dokumenta

Za početak je potrebno napomenuti da prilikom generisanja pdf dokumenta na osnovu Isabelle/HOL fajla, neće biti prikazani dvostruki navodnici koji se koriste prilikom zapisivanja formulacije lema i teorema. Ovo je nešto što se može primetiti i u ostalim Isabelle/HOL tutorijalima.

Iako je Isabelle trenutno usmeren ka interaktivnom pisanju koda, prevođenje u pdf dokument se izvršava u konzoli. Alati koji se koriste u ovu svrhu su *isabelle mkroot* i *isabelle build*. Više detalja o njima se može naći u dokumentima `tutorial.pdf` i `system.pdf`.

Alat *mkroot* se koristi za kreiranje potrebnih foldera i fajlova koji se koriste za lakšu manipulaciju sa više različitih Iabelle fajlova. On se pokreće samo jednom. Alat *build* služi kreiranju samog pdf fajla i pokreće se nakon svake promene nad Isabelle/HOL fajlovima ili nad samom strukturom generisanog tex fajla. Prvi korak je pokretanje narednih dveju naredbi (gde umesto *isabelle* navedete tačnu putanju (npr. */Downloads/Isabelle2020/bin/isabelle*).):

```
isabelle mkroot MySession
isabelle build -D MySession
```

Rezultat pokretanja prve naredbe će biti kreiranje foldera `MySession` sa podfolderima `output`, `document` i fajlom `ROOT`. Ako sada pokrenemo odmah i drugu naredbu trebalo bi da u folderu `output` dobijete fajl `document.pdf` sa nekim osnovnim sadržajem.

Sada možemo dodavati svoje thy fajlove u folder MySession. Nakon toga potrebno je dodati ih u fajl `ROOT` nakon reda u kome piše *theories*. Navode se samo imena fajlova, bez thy ekstenzije. Ako je potrebno, pre sledećeg prevođenja može se srediti i generisani tex dokument koji se nalazi u folderu `document` (na primer najčešće će biti potrebno promeniti ime autora, dodati neke pakete za prevođenje generisanog tex dokumenta i slično).

Neke od ugrađenih Isabelle/HOL teorija zahtevaju da se uključe u ROOT fajl nakon ključne reči *session*, na primer ako želimo da koristimo teorije *HOL−Library.Permutation* (za permutacije), *Tutorial.Pairs* (za rad sa uređenim parovima) i *HOL−ex.Sqrt* (za računanje korena realnog broja) naveli bi sledeću naredbu u okviru ROOT fajla:

```
sessions "HOL-Library" "Tutorial" "HOL-ex"
```

Ako želimo da fajlove organizujemo u **dodatnim folderima** (u okviru foldera *MySession*, potrebno je imena tih foldera navesti u ROOT fajlu naredbom `directories directory_name`.

Da bi se preveli u pdf fajlovi u kojima je dozvoljeno ostaviti određene teoreme (ili delove teorema) nedokazanim, to postižemo korišćenjem naredbe `sorry`. U tom slučaju moramo koristiti `quick_and_dirty` mode ili u samom fajlu (kao što je već ranije rađeno) ili u ROOT fajlu navesti naredbu `theories [quick_and_dirty]` ispred takvih fajlova. Ako ne želimo da menjamo same fajlove možemo i navesti prilikom prevođenja kao dodatnu opciju:

```
isabelle build -o quick_and_dirty ...
isabelle build -o quick_and_dirty=true ... # podrazumevano je true, pa je ovo isto kao prethodno
isabelle build -o quick_and_dirty=false ...
```

U nekim situacijama kada *sledgehammer* predloži korišćenje *smt* rešavača, može da se desi da naredba ne uspe da se izvrši prilikom prevođenja dokumenta pa se može ograničiti njihovo vreme izvršavanja naredbom:

```
declare [[smt_timeout = 20]].
```

A.2 Ispis teksta u Isabelle/HOL

Prilikom generisanja pdf dokumenata komentari koje smo do sada koristili, odnosno tekst između (∗ i ∗) neće biti prikazan. Da bi se dobili komentari koda koji se vide u izlaznom pdf dokumentu potrebno je koristiti naredbu `\<comment>` koja se prikazuje kao —.

Umesto ASCII simbola, naredni Isabelle/HOL simboli se koriste kao oznake za liste: `\ < ^ item>` za nenumerisane liste, `\ < ^ enum>` za numerisane liste, `\ < ^ descr>` za opisne liste (razmake obrisati prilikom kucanja u Isabelle/HOL i biće prikazan odgovarajući simbol).

A.2.1 Ispis teorema u Isabelle/HOL

Komanda `@{thm conjI}` ispisuje u Isabelle/HOL teoremu na sledeći način $?P \wedge ?Q \Longrightarrow ?P \wedge ?Q$. Postoji nekoliko opcija za ispisivanje teorema, na primer naredba @{*thm*[*display*,*mode*=*latex-sum*] *sum-Suc-diff* [*no-vars*]} koja ispisuje:

$m \leq Suc\ n \Longrightarrow (\sum_{i=m}^{n} f\ (Suc\ i) - f\ i) = f\ (Suc\ n) - f\ m$

Ako prilikom ispisa teoreme *conjI* ne želimo da se prikazuju znakovi pitanja možemo iskoristiti opciju *no-vars*: `@{thm conjI [no_vars]}` i dobićemo $P \wedge Q \Longrightarrow P \wedge Q$.

Pravila prirodne dedukcije u Isabelle/HOL U Isabelle/HOL se mogu koristiti standardna tex-ovska pravila za zapisivanje pravila prirodne dedukcije: [1]

$$\frac{P \wedge Q}{P} \qquad \frac{P \wedge Q}{Q}$$

a mogu se koristiti i naredbe u okviru teorija *LaTeXsugar* i *OptionalSugar* koje se uključuju uz pomoć direktne putanje naredbama:

~/*Downloads*/*Isabelle2020*/*src*/*HOL*/*Library*/*LaTeXsugar*

~/*Downloads*/*Isabelle2020*/*src*/*HOL*/*Library*/*OptionalSugar*.

Pored ovoga potrebno je uključiti i Didier Rèmy's `mathpartir` paket u *root.tex* fajl.

Naredba `@{thm[mode=Proof] conjE [no_vars]}` proizvodi sledeći ispis: $[\![P \wedge Q;\ P \wedge Q \Longrightarrow R]\!] \Longrightarrow R$ čak i u sred rečenice. Ovaj ispis odgovara interpretaciji u okviru Isabelle/HOL.

Naredba `@{thm[mode=Rule] conjE [no_vars]}` takođe može da proizvede formulu unutar rečenice ali nije praktično tako raditi. Bolje je koristiti ovu naredbu za nezavisan ispis.

$$\frac{P \wedge Q \qquad \dfrac{P \wedge Q}{R}}{R}$$

Može se dodati i naziv pravila:

$$\frac{?P \wedge ?Q}{?P \wedge ?Q}\ \textsc{conjI}$$

[1] https://www.logicmatters.net/resources/ndexamples/proofsty.html

A.2.2 Ubacivanje dela Isabelle koda u nezavisan tex dokument

Delovi postojeće Isabelle formalizacije, teorije, definicije i dokazi mogu se izdvojiti i iskoristiti u nezavisnom Latex dokumentu. Odgovarajući tex kod se mora prvo izgenerisati uz pomoć Isabelle (na način opisan u prethodnom poglavlju). Deo koda koji nas zanima se onda smešta u rezultujući tex fajl između naredbi $\backslash begin\{isabelle\}$ i $\backslash end\{isabelle\}$ ili $\backslash isa\{...\}$.

end