UNIVERZITET U BEOGRADU
MATEMATIČKI FAKULTET

Nikola Katić

SAMOKOMPILIRAJUĆI PREVODILAC U OKVIRU INTERAKTIVNOG DOKAZIVAČA TEOREMA

master rad

Beograd, 2022.

Mentor:

prof. dr Filip Marić, vanredni profesor
Univerzitet u Beogradu, Matematički fakultet

Članovi komisije:

prof. dr Milena Vujošević Janičić, vanredni profesor
Univerzitet u Beogradu, Matematički fakultet

dr Sana Stojanović Đurđević, docent
Univerzitet u Beogradu, Matematički fakultet

Datum odbrane: _____

Naslov master rada: Samokompilirajući prevodilac u okviru interaktivnog dokazivača teorema

Rezime: Ovim radom je približen problem verifikacije prevodioca, modelovanjem jezika u dokazivaču teorema *Coq*. Samokompilirajući prevodilac je onaj koji je u stanju da prevede sopstveni izvorni kôd. Projektom *CakeML* je na jedan neuobičajen način dobijen samokompilator – u okviru interaktivnog dokazivača teorema (IDT). Jedan od autora tog projekta je u radu [14] približio korišćene koncepte, a cilj ovog rada je njihov prikaz. U sklopu toga će u ovom master radu biti analiziran specifičan pristup prevođenju koda, kojim se zadržavaju semantička svojstva jezika sa kojeg se prevodi. Pristup verifikaciji prevodilaca pomoću IDT-a podrazumeva opis osobina i ponašanja ulaznog i ciljnog jezika. Implementirane su apstraktne sintakse jednostavnih jezika koji podsećaju na Lisp i stek mašinu. Uz dodatne pretpostavke o obliku izraza, realizovano je i prevođenje sa jednog na drugi jezik.

Ključne reči: verifikacija, interaktivno dokazivanje teorema, prevodioci

Sadržaj

Glava 1

Uvod

Kompilator (ili prevodilac) je program koji je u stanju da dati tekst programa napisanog u *ulaznom* (engl. *source*) jeziku transformiše u program napisan u nekom *ciljnom* (engl. *target*) jeziku. Prvi viši programski jezici su nastali sa pojavom ideje da se podigne nivo apstrakcije pri zadavanju instrukcija računaru. Zamisao načina funkcionisanja viših jezika je prvi put dobila i praktičnu formu, onog trenutka kada su napisani prvi prevodioci za njih. Da bi se napisao kompilator za neki viši programski jezik, moralo se pribeći korišćenju asemblera ili mašinskog koda. Specijalan slučaj u kojem kompilator prevodi sopstveni kôd se naziva *samokompilacijom*. Umesto da *ceo* kompilator bude implementiran u asembleru, moguće međurešenje podrazumeva pravljenje kompilatora u asembleru za neki fragment ulaznog jezika. Poželjno je da taj fragment bude minimalan, kako bi implementacija njegovog prevodioca bila olakšana. Pritom je neophodno da se odabranim podskupom mogu opisati svi preostali konstrukti jezika. Zatim se tako dobijenim kompilatorom prevodi implementacija prevodioca za ceo jezik, napisana u podržanom fragmentu.

Problem samokompilacije u interaktivnom dokazivaču teorema (IDT) koji će u nastavku biti razrađen može se doživeti kao spoj tri veće oblasti računarstva. Prevođenje programskih jezika je jedna od njih, međutim, sâm postupak neće biti detaljno prikazan u duhu uobičajene formulacije jer ne predstavlja preduslov za opisan pristup verifikaciji. Drugi važan aspekt je formalna verifikacija softvera, odnosno dobijanje potvrde da su datim računarskim kodom zadovoljeni određeni kriterijumi. Treći važan činilac u rešavanju ovog problema je oblast interaktivnog dokazivanja teorema, čije se tehnike mogu iskoristiti kao okvir za verifikaciju softvera. Prikaz upotrebe tehnika pomenutih oblasti, iako značajno pojednostavljen u ovom radu, može služiti kao smernica pri rešavanju srodnih problema.

Ovim radom dat je širi kontekst koji se tiče verifikacije prevodilaca korišćenjem IDT-a. U tu svrhu, prikazane su i osnovne mogućnosti dokazivača *Coq*. Približena je ideja samokompilacije i generisanja koda automatskim postupkom dokazivanja. Takođe, praktično je predstavljen način modelovanja jezika unutar dokazivača. U glavi 2, počevši od osnovnih pojmova i prvih primera verifikovanih prevodilaca, postepeno je davana motivacija za poduhvat dokazivanja korektnosti kompilatora i navedena su dva glavna savremena primera verifikovanih kompilatora u okviru IDT-a, od kojih je jedan samokompilirajući prevodilac. Minimalni jezik kojim je moguće demonstrirati osnovne korake pri modelovanju jezika u *Coq*-u, dat je u glavi 3. U glavi 4 prikazani su ključni koncepti iz rada o verifikovanoj samokompilaciji [14], autora Magnusa Mirina (*Magnus O. Myreen, Chalmers University of Technology*), kao i detaljna analiza ideja na kojima je zasnovan poznat primer verifikovanog samokompilatora. U glavi 5 je dat model ulaznog jezika koji se naslanja na prethodno implementirani jezik iz glave 3, ali obogaćen idejama iz glave o samokompilaciji 4, zajedno sa primitivnim ciljnim jezikom i svedenom podrškom za automatizovano prevođenje sa jezika *Coq*-a na apstraktnu sintaksu ulaznog jezika. Osvrt na temu i zaključak napisani su u poslednjoj glavi rada 6.

Glava 2

Kompilatori i verifikacija softvera

U tekstu koji sledi, precizirani su zadatak i željena svojstva kompilatora, a uvedeni su i neki od osnovnih pojmova koji se koriste pri njihovoj konstrukciji. Dat je pogled na verifikaciju kompilatora korišćenjem IDT-a, kao i kratak pregled nekih korišćenih elemenata sistema *Coq*. Spomenuti su prvi primeri, kako verifikovanog algoritma prevođenja, tako i mehanički (pomoću računara) dokazanog korektnog izvršavanja kompilatora i samokompilatora.

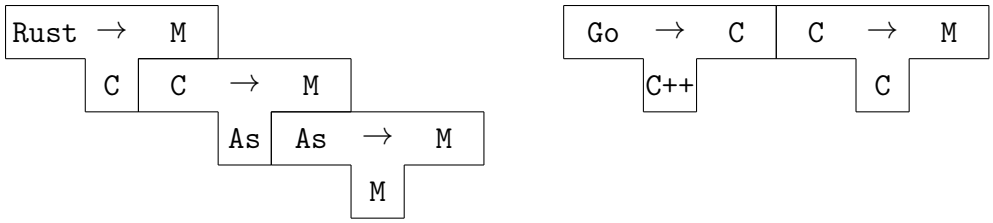
2.1 Proces kompilacije

Osnovni zadatak kompilatora je da prevede program sa ulaznog na ciljni jezik, tako da prevedeni kôd u ciljnom jeziku zadrži svojstva koja poseduje prvobitni program, napisan u ulaznom jeziku. Najčešći slučaj je da se sa jezika višeg nivoa vrši prevođenje u jezik niskog nivoa poput asemblera, a u krajnjem slučaju to je mašinski kôd, kao jezik direktne komunikacije sa hardverom za koji je prevodilac namenjen. Jezik pomoću kojeg je implementiran prevodilac se naziva *implementacijskim* (engl. *implementation language*). U fazi donošenja odluka o dizajnu prevodioca i prilikom odabira ciljnog jezika, jedno njegovo važno svojstvo je da bude *izvršiv* ili da se veoma jednostavno prevodi u izvršiv oblik. To znači da ga je moguće pokrenuti kao izvršnu datoteku, direktno – kada je ciljni jezik mašinski, ili korišćenjem nekog dodatnog mehanizma za izvršavanje.

Pojmovi ulaznog, ciljnog i implementacijskog jezika su u svetu konstrukcije kompilatora uobičajeni – jasno su istaknuti čestim načinom slikovnog predstavljanja kompilatora, takozvanim *T-dijagramima* (od engl. sintagme *tombstone diagrams*). Njihovim nadovezivanjem mogu se dobiti oblici kao na slici 2.1.

Sa leve strane slike 2.1 se nalazi prevodilac napisan u C-u, koji programe u *Rust*-u prevodi u izvršive programe na mašinskom kodu M. Taj kompilator je nastao korišćenjem kompilatora za programe u C-u (napisanog u asembleru As), kojim se dobija izvršiv kôd. Analogno prethodnom, poslednji kompilator napisan je za asembler As (u mašinskom kodu M) i obavlja prevođenje takođe u mašinski kôd M. Ulančavanje T-dijagrama, se u tom smislu može iskoristiti zarad predstavljanja istorije konstrukcije nekog postojećeg kompilatora, ili kako bi se skicirao plan konstruisanja.

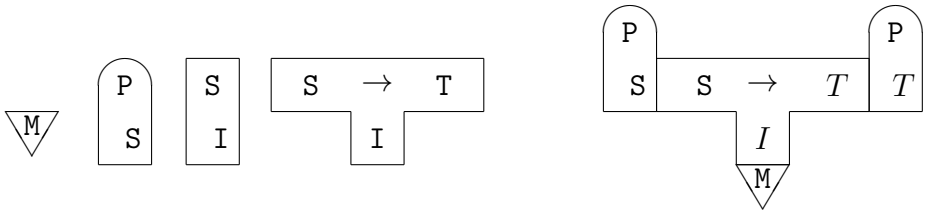
Sa desne strane slike 2.1 prikazan je drugačiji tip veze. Prvi kompilator obavlja prevođenje iz Go-a u C, a taj izlaz se prosleđuje drugom kompilatoru za programski jezik C. Poslednje pomenuti kompilator se naziva *samokompilirajući* jer je napisan u istom jeziku sa kojeg prevodi. *Samokompilatori* se, dakle, mogu definisati kao prevodioci čiji su ulazni i implementacijski jezik isti. Treba primetiti da odgovarajući T-dijagrami ne bi odavali potpunu sliku o svim prevodiocima koji su korišćeni. Kako C nije sam po sebi izvršiv jezik (kao što je to mašinski), taj prevodilac je takođe morao u nekom trenutku da bude napisan pomoću mašinskog ili nekog drugog jezika za koji postoji način da se izvrši.



Slika 2.1: Ilustrativni primeri T-dijagrama.

U kraćim crtama, početkom šezdesetih godina prošlog veka, Harvi Bretmen (*Harvey Bratman*) je došao na ideju T-dijagrama, kao unapređenja već postojećih dijagrama *UNCOL*[1]. Decenijama nakon toga, davani su predlozi alternativnih dijagrama, kako bi se prevazišle pojedine manjkavosti sa višeznačnošću. Jedan predlog unapređenja je dat u vidu dijagrama *Earley–Sturgis* prikazanih na slici 2.2. Iako dijagrami *Earley–Sturgis* predstavljaju potpuniju informaciju, u praksi se i dalje koriste T-dijagrami. Detaljniji opis, istorijski osvrt i poređenje sa drugim tehnikama se može naći u radu u kome se predlažu dalja unapređenja T-dijagrama [9].

[1]Prvobitno kao ideja stvaranja univerzalne međureprezentacije programskog koda (iz 1950-tih).



Slika 2.2: Primer dijagrama *Earley–Sturgis*. Prva četiri elementa sa leve strane su gradivni blokovi koji redom predstavljaju: mašinu M, program P napisan u jeziku S, interpretator sa jezika S, napisan u jeziku I, T-dijagram kompilatora napisanog u I, kojim se prevodi sa jezika S na T. U skladu sa tim, može se zaključiti značenje dijagrama *Earley–Sturgis* sa desne strane: program P napisan u jeziku S se prevodi na mašini M, kompilatorom napisanim u jeziku I (datim u obliku izvršivog koda mašine M), a rezultat koji se dobija je program P napisan u jeziku T.

2.2 Verifikacija kompilatora

U nastavku su navedeni motivišući primeri koji ukazuju na značaj verifikacije kompilatora. Takođe, približeni su različiti načini tumačenja značenja programa, što predstavlja preduslov za formiranje tvrđenja o njegovoj korektnosti.

Svrha prevodioca, navedena na početku poglavlja 2.1 o procesu kompilacije, nije bila dovoljno precizna. Moguće je zapitati se, koja su to svojstva koja je važno očuvati? Kako oceniti da li su ona zadržana? Pitanjima tog tipa se bavi oblast *verifikacije softvera*, a mnoštvo odgovora bi se moglo ponuditi, u zavisnosti od okvira u kojima se problem razmatra i od odluka koje se donose u skladu sa odabranim aspektima od važnosti.

Pojmom *verifikacije softvera* obuhvaćene su raznovrsne grupe tehnika. Odabir formalnog načina kojim bi se ispravnost nekog programa dokazala predstavlja početni korak i njime mogu biti uslovljene mogućnosti u daljim fazama postupka dokazivanja željenih osobina. Verifikacija kompilatora ima svojih specifičnosti. Cilj verifikacije nekog konkretnog softvera usmeren je ka dokazivanju određenih njegovih svojstava, odnosno ponašanja. Kada se dokazuje korektnost kompilatora (kao klase programa), tada se mogu uočiti pojedine sličnosti u pogledu ciljeva verifikacije. S obzirom na to da je ponašanje koje kompilator implementira *prevođenje koda*, za očekivati je da je upravo *prevođenje* ono što je važno da bude ispravno. Prevođenje je ispravno ukoliko čuva osobine od interesa, a da bi se ta svojstva programa opisala i dokazala postupkom od poverenja, potrebna je njihova precizna formulacija.

Prvi korak ka definisanju svojstava je pridodavanje značenja konstruktima jezika, što se naziva *semantikom* programskih jezika. *Neformalna semantika* podrazumeva smernice, iskazane prirodnim jezikom, kojima se programer rukovodi u praksi, prilikom razvijanja koda. Može biti data u vidu uputstva namenjenog programerima, ili stečena kroz iskustvo, korišćenjem jezika (ili, što je najverovatniji slučaj – oba). Nezavisno od načina usvajanja, neformalna semantika, iako korisna za razmenu informacija među ljudima, ostaje u domenu *nedovoljno preciznog* u kontekstu formalnog rezonovanja. Kada bi se za pisanje prevodilaca koristila neformalna semantika, u situacijama kada instrukcije o očekivanom ponašanju nisu dovoljno precizne, različiti prevodioci bi mogli ispoljavati različito ponašanje [12].

Formalna semantika namenjena je rezonovanju pomoću programa, za razliku od neformalne koja je namenjena ljudima [12]. Kada postoji formalna semantika onda se zadatak verifikacije kompilatora može preformulisati u zadatak očuvanja semantike programa nakon prevođenja. U poglavlju koje se odnosi na definisanje jezika, biće više reči o tipovima formalne semantike kroz konkretne primere.

Prvim dokazom tačnosti kompilatora se smatra rad Džona Makartija (*John McCarthy*) i Džejmsa Pejntera (*James Painter*) iz 1967. godine [11]. U radu je opisan *algoritam prevođenja* aritmetičkih izraza, čiju su korektnost zatim i dokazali. Prvi dokaz koji je i mehanički bilo moguće proveriti, objavljen je 1972. godine [13]. Izvorni jezik je bio imperativni, sa osnovnim konstruktima poput operatora dodele, grananja i *while*-petlje, a ciljni jezik je bio asemblerski. Sâm dokaz je dat u sistemu *LCF* (engl. *Logic for Computable Functions*), uz ideje za automatizovanje postupka dokazivanja.

U narednom delu se navodi eksperiment kojim je ukazano na prednost verifikovanog kompilatora u odnosu na neverifikovane kompilatore. *Csmith* je alat kojim se generišu nasumični programi u C-u u skladu sa *standardom C99*. Osnovna namena alata *Csmith* je pronalaženje propusta u kompilatorima jezika C i drugom softveru namenjenom nekoj vrsti obrade koda napisanog u C-u. U radu iz 2011. godine, kojim je urađena sveobuhvatna analiza dostupnih kompilatora C-a [23], pokazano je da je svaki posedovao propuste koji su dovodili do iznenadnog prestanka rada kompiliranih programa i propuste u smislu pogrešnih izračunavanja. Naziv korišćene tehnike je *nasumično testiranje poređenjem* (engl. *differential fuzz testing*). Generisani validni programi su se prosleđivali kao ulazi za više različitih kompilatora. Pretpostavka je da svi kompilatori treba da proizvode programe koji daju iste rezultate. *Kontrolnom sumom* (engl. *checksum*) su se enkodirala stanja programa nakon izvršavanja

nekog izračunavanja. Većinski rezultat je proglašavan ispravnim, a kompilatorima koji su davali programe sa drugačijim izlazima su beležene greške.

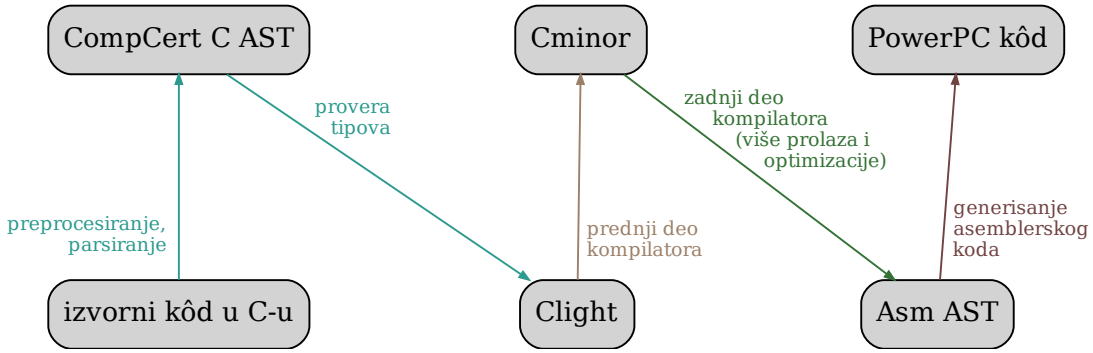
Pri vršenju tako nasumičnog generisanja, ipak je bilo potrebno voditi računa o nekim situacijama. Standardom C99 je ostavljeno blizu 200 slučajeva u kojima je *ponašanje nedefinisano*[2] (engl. *undefined behavior*), posle kojih je uzaludno dalje razmatrati semantiku. Kod ovog eksperimenta, podjednako važnu situaciju su predstavljali i slučajevi u kojima je *način ponašanja prepušten implementaciji kompilatora* (engl. *unspecified behaviors*). Njima je moguće dobiti, primera radi, različite redoslede nekih izračunavanja, što uvodi određeni nedeterminizam koji se ne uklapa u osnovnu pretpostavku eksperimenta. Zato su autori pribegavali statičkoj analizi, i dodavanju provera u toku izvršavanja programa.

CompCert C

Drugim, nezavisnim projektom, nazvanim *CompCert*, verifikovan je kompilator *CompCert C* u dokazivaču *Coq*. Taj kompilator prevodi jezik *Clight*, koji predstavlja značajan podskup programskog jezika C, na asembler za arhitekturu *PowerPC* [11]. U vreme kada se pomenuti eksperiment sa korišćenjem alata *Csmith* odvijao, samo je *srednji deo* kompilatora *CompCert* bio verifikovan. Srednji deo kompilatora vrši optimizacije na nivou međureprezentacije. Od svih razmatranih kompilatora, jedini kojem u tom delu nisu pronađena pogrešna izračunavanja je *CompCert* [23]. U međuvremenu, *CompCert* je postao prvi prevodilac C-a (na asembler) koji je *skoro u potpunosti* verifikovan. Poslednjom procenom (iz juna 2022. godine) to je 90% algoritma za kompilaciju [10]. Njime su podržane optimizacije koda, a performanse prevedenog koda su reda veličine performansi programa prevedenih korišćenjem neverifikovanog kompilatora *gcc* [1].

Kompilacija je realizovana sa 20 prolaza tokom kojih se vrši prevođenje u 11 međujezika. Jedan od njih je imperativni jezik niskog nivoa *Cminor*, prikazan na slici 2.3 kojom je dat uprošćeni pregled delova kompilacije. Delovi kompilacije su grupisani na isti način kao u dokumentaciji [1]. To su redom: parsiranje, prednji deo kompilatora, zadnji deo kompilatora i generisanje asemblerskog koda. Srednji deo kompilatora je obuhvaćen delom koji se u dokumentaciji kompilatora *CompCert* označava kao zadnji deo.

[2]Na primer, pokušaj oslobađanja već oslobođene dinamički alocirane memorije.



Slika 2.3: Uprošćeni dijagram strukture kompilatora *CompCert*.

Semantika svakog od međujezika formalizovana je u sistemu *Coq*, u kome su implementirani i prolazi, sa ciljem dokazivanja korektnosti *svih* faza prevođenja. Rezultat projekta *CompCert* je softver koji je praktično upotrebljiv. To potvrđuju primeri korišćenja u nuklearnoj (*MTU Friedrichshafen*) i avioindustriji (*Airbus France*) [1]. *CakeMl*, *Pilsner*, *CompCertTSO*, *Cogent* i *Fiat* su još neki od istaknutih primera verifikovanih kompilatora [15].

2.3 Samokompilacija

U ovom poglavlju je dat primer projekta kojim je na specifičan način dobijen verifikovani samokompilator. Takođe, naveden je slučaj kojim se dovodi u pitanje pouzdanost uobičajenog postupka postizanja samokompilacije.

Ako se razmatraju kompilatori (a ne interpretatori ili drugi oblici prevođenja) koji su u današnje vreme u upotrebi, nije retkost da su napisani u istom jeziku za koji su namenjeni. Potencijalno veći broj prethodnih verzija mogao je biti napisan u istom jeziku i dokle god postoji javni izvorni kôd prevodioca, moguć je uvid u njegov način rada.

Ken Tompson[3] je istakao zanimljivu poentu u kratkom radu izlaganom 1984. godine [22]. Kao jedan od autora operativnog sistema *UNIX*, dao je primer koji se ticao kompilatora za jezik C. Naime, istakao je pitanje koliko se može verovati, u bilo kojoj fazi korišćenim, izvršnim datotekama kompilatora. Čak i kada postoji izvorni kôd samokompilatora za C, to nije dovoljno za otklanjanje svih sumnji o eventualnom zlonamernom delovanju izvršnog oblika te datoteke. Posedovanje izvornog koda

[3] *Ken Thompson*, američki informatičar, dobitnik *Tjuringove nagrade* 1983. godine.

svih verzija kompilatora, ili slikovito, mogućnost razvijanja ulančanog T-dijagrama sve do kompilatora implementiranog u jeziku najnižeg nivoa jedina je prava garancija. Hipotetički primer takvog T-dijagrama može se videti u levom delu slike 2.1 na strani 4.

Zlonameran programer bi, znajući šta je čest predmet prevođenja (u ovom slučaju operativni sistem *UNIX*) mogao da u izvornom kodu uoči bezbednosno važan deo[4] i da taj kôd izmeni po svojoj potrebi. U svrhu smanjivanja vidljivosti odstupanja od ponašanja netaknutog kompilatora, kada god se kompilira neki drugi program, razlike u prevođenju (u odnosu na netaknut kompilator) ne bi bilo. Nemogućnošću analize toliko velikog koda, koji se dobija prevođenjem kompilatora, a eventualnim uveravanjem u prevode nekih manjih programa, mogla bi se načiniti pretpostavka da je kompilator bezbedan. Vodeći se time, kompilacijom naredne verzije prevodioca, sa izvornim kodom bez umetnutog zlonamernog dela, dobila bi se nova izvršna verzija kojoj bi se moglo verovati. Načinjena pretpostavka nije nužno dobra budući da bi se na sličan način kao i kod prepoznavanja bezbednosno-osetljivog dela, kôd naredne verzije *kompilatora* mogao takođe prepoznati. Nakon uočavanja da je predmet prevođenja kompilator, u rekurzivnom maniru, umetanjem celog zlonamernog dela koda, sigurnosni propust bi se propagirao u svakoj narednoj izvršnoj verziji prevodioca (kada se prevodi *UNIX*). Diskutabilno je koliko je ovakav scenario realističan, s obzirom da bi zlonameran deo koda morao biti dovoljno sofisticiran da može da identifikuje delove koda koji su podložni promenama u narednim verzijama kompilatora (i u pomenutom slučaju i operativnog sistema *UNIX*). U kontekstu dolaženja do samokompilirajućeg prevodioca, ali i korišćenja programa inače, takva tema ostavlja prostor za preispitivanje.

Jedna od prednosti konstruisanja samokompilirajućeg prevodioca, u slučaju dizajniranja sopstvenog jezika, je njihova paralelna evolucija. Kako se razvijaju nove funkcionalnosti u jeziku, tako se njima nadograđuje i prevodilac (iz naredne iteracije). Netrivijalan softver poput kompilatora predstavlja svojevrsni test za sâm jezik. Samokompilatorom se u tom smislu šalje dobra poruka potencijalnim korisnicima o mogućnostima jezika.

[4]Kao primer u originalnom radu je navedena komanda *login*.

CakeML

Zapažen rezultat verifikacije prevodioca postignut je projektom *CakeML*, gde ne samo da je u potpunosti verifikovan kompilator, već je u interaktivnom dokazivaču teorema napravljen verifikovani samokompilator. Magnus Mirin, autor rada [14], ujedno je i jedan od tvoraca samokompilatora *CakeML* i u [14] je prikazao kako je to sprovedeno, ali na manjem primeru. Konceptualni prikaz tog rada će biti dat u ovom master radu.

CakeML predstavlja prvi samokompilirajući formalno verifikovani kompilator. Samokompilator je dobijen mehanizmom za generisanje koda implementiranim u dokazivaču teorema *HOL4*. U metajeziku dokazivača *HOL4* su implementirani delovi za leksiranje, parsiranje, zaključivanje tipova i generisanje koda. Korišćenjem sistema *CakeML* je za brojne aplikacije dokazano da su tačne do mašinskog koda, a za neke i do hardvera na kome rade. Pojednostavljena struktura kompilatora prikazana je na slici 2.4. Postupak se izvršava koristeći 8 međujezika, za koje se vrši po nekoliko prolaza. Sistem omogućava generisanje koda u *SML*-u, podskupu jezika *ML* (engl. *Standard ML*), *OCaml* ili *Haskell*. Opšti cilj je da se dokaže očuvanje osobina nakon prevođenja od izvornog do mašinskog koda, ali tako da u formulaciji krajnje teoreme o korektnosti ne bude funkcije kompilacije, već konkretnog izvršivog koda dobijenog kao prevod [17].

Slika 2.4: Uprošćeni dijagram strukture samokompilatora *CakeML*.

2.4 IDT u verifikaciji softvera

U kontekstu verifikacije prevodilaca, u ovom poglavlju približen je pojam interaktivnog dokazivača teorema. Da bi dokaz bio formalan, neophodan je odgovarajući okvir u kojem je moguće precizno zaključivati. Davanje tvrdnji i njihovo dokazivanje

se vrši na *metajeziku*. U praktičnom delu ovog rada, metajezik koji će biti korišćen je jezik za specifikaciju interaktivnog dokazivača teorema *Coq*, nazvan *Gallina*.

Dokazivanje se može sprovesti krećući se u dva različita smera. Kada se korišćenjem nekog skupa *pravila* i nekog skupa *pretpostavki*, ta pravila (postepeno) primenjuju na pretpostavke, takav način zaključivanja se naziva *rezonovanjem unapred*. Time se dobijaju novi zaključci kojima se može proširiti skup pretpostavki. Moguće je kretati se i u suprotnom smeru, a takav stil zaključivanja se naziva *rezonovanjem unazad* (engl. *backward reasoning*). U tom slučaju se transformiše *cilj* (tvrdnja koju treba dokazati) na jedan ili više *potciljeva* koje je *dovoljno* dokazati [18]. Transformacije kojima se cilj razlaže na jednostavnije ciljeve, zovu se *taktike* (engl. *tactics*).

Pomenuti sistem *LCF*, u kojem je sproveden prvi mehanički dokaz kompilatora, doneo je nove značajne koncepte. *LCF* je prvi implementirani sistem koji je posedovao mogućnost dokazivanja unazad i sem taktika, uveden je i pojam *taktika višeg reda* (engl. *tacticals*) [7]. Taktikama višeg reda je moguće izgrađivati strategije dokazivanja (kombinovanjem taktika), čime se proces dokazivanja može olakšati ili ubrzati.

Prilikom korišćenja određenog sistema, dobro je znati koji su elementi tog sistema za koje se *veruje da su ispravni* (*TCB*, od engl. *trusted code/computing base*), a čijom neispravnošću poverenje u ceo sistem biva poljuljano. Kao i u pomenutom slučaju pravljenja kompilatora, gde postoji oslanjanje na neku već postojeću izvršnu datoteku, tako i u slučaju IDT, prirodno je zapitati se koje su garancije da je zaključak koji se dobija pouzdan. Sama teorijska pozadina među različitim dokazivačima se može bitno razlikovati, te je teško davati ocene o pouzdanosti interaktivnih dokazivača uopšteno. Sveobuhvatan pregled bitnih aspekata dokazivača postoji u poglavlju *„Reliability of proof checking"* rada [7]. Ono što im je svima zajedničko je cilj postojanja velikih garancija pouzdanosti njihovih zaključaka, što se može postići minimalnim *TCB*-om. Postavljanje jasne granice između *TCB* dela u kojem bi postojanje propusta ugrozilo pouzdanost sistema i ostalih delova (u kojima je u tom smislu propust moguć) je dobar način za razumevanje koliko je neki sistem verodostojan [2].

Proveravač dokaza je program koji proverava ispravnost datog dokaza. U slučaju sistema *Coq*, sâm proveravač se nalazi u *TCB*-u[5], zajedno sa svim propratnim

[5]Teoretska osnova dokazivača *Coq* i korespondencija između *tipova* i *dokaza* dozvoljava da proveravač bude relativno mali program [18].

alatima i tehnikama kojima je napravljen [2]. Za razliku od proveravača, implementirani mehanizam za taktike se ne nalazi u *TCB*-u. *Coq* je napisan korišćenjem jezika *OCaml*, pa je i korišćeni kompilator za *OCaml*, kao i operativni sistem na kojem se prevodilo, zajedno sa hardverom na kojem se prevodilac izvršavao, takođe u *TCB*-u. Od nekog trenutka realna briga o stopostotnoj sigurnosti u ispravnost prestaje. U suprotnom, predmet brige postaje sve teže uhvatljiv. Prostor u kojem mogu da se traže dodatne garancije ispravnosti deluje neograničen usled nesagledivosti liste *svih* mogućih procesa od kojih rad nekog softvera zavisi [2].

Kada je reč o programskim jezicima i formalnom rezonovanju o njima, ukoliko je to zadovoljavajuće, umesto da se formalizuje ceo jezik, moguće je opisati samo određene delove od značaja i rezonovati o njima. U tom smislu, moguće je enkodirati samo pojedine funkcije u terminima logike metajezika i dokazivati svojstva o njima. Primera radi, neka je $kvb(x, y) = x^2 + 2xy + y^2$ funkcija u jeziku koji se formalizuje i za koju treba da se dokažu određene osobine. Prvi pristup je da se cela sintaksa jezika najpre enkodira u jeziku IDT, da se sintaksi pridruži odgovarajuće značenje, pa tek onda obavi dokazivanje željenih svojstava te funkcije. O takvom pristupu, koji se naziva *duboko utapanje* (engl. *deep embedding*), biće više reči u nastavku. Radi sticanja uvida, primer iskorišćavanja takvog pristupa je prikazan u kodu 2.5.

```
Op Add [Op Add [Op Mult [Var x; Var x] ;
  Op Mult [Op Mult [Const 2 ; Var x] ; Var y]] ; Op Mult [Var y ; Var y]]
```

Kôd 2.5: Duboko utapanje funkcije $kvb(x, y) = x^2 + 2xy + y^2$.

Ista funkcija se može i direktno opisati, korišćenjem *isključivo* već postojećih konstrukata jezika *Gallina*:

```
Definition kvb := (fun x y  => (x*x + 2*x*y + y*y)).
```

Već na čisto vizuelnom nivou, deluje kao da je drugi izraz jednostavnije koristiti pri dokazivanju. Razlog tome je nepostojanje dodatnog sloja između jezika dokazivača i onoga što je predmet dokazivanja. Ipak, postoje poteškoće do kojih se dolazi takvim (direktnim) prepisivanjem funkcija na metajezik, na primer, otežano je rezonovanje o nekim svojstvima programskog jezika koji se enkodira. Nasuprot tome, kada se koristi prvi pristup, sintaksa jezika se induktivno definiše, čime zaključivanje o svojstvima svih mogućih izraza tog tipa (odnosno svim sintaksički validnim izrazima jezika) postaje prirodno – uz pomoć indukcije [2].

Drugi pomenuti pristup se naziva *plitkim utapanjem* (engl. *shallow embedding*), a prvi, u kojem se jezik specijalno definiše i interpretira se naziva dubokim utapanjem. Treća mogućnost je *mešovito utapanje* (engl. *mixed embedding*), koje predstavlja kompromis prethodna dva pristupa. Razmatranje slučajeva u kojima je taj pristup pogodniji je dato u radu o mešovitom utapanju [3].

Ukratko, jedan primer odstupanja od dubokog utapanja se dobija neznatnom izmenom definicije uobičajenog *let*-izraza. S obzirom na to da se radi o dubokom utapanju, `let` je predstavljen konstruktorom induktivnog tipa, čiji je potpis:

```
Let : vname -> exp -> exp -> exp
```

Značenje potpisa ovakvog pravila je da se kao prvi argument uzima identifikator tipa `vname`. Promenljivoj koju identifikator predstavlja se dodeljuje vrednost izraza prosleđenog kao drugi argument, a zatim se ta promenljiva (sa novom vrednošću) koristi u izrazu koji je prosleđen kao treći argument i koji se obično naziva *telom* naredbe *let*. Funkcija vraća vrednost izračunavanja izraza, što odgovara poslednjem `exp` iz potpisa.

Umesto ovakvog načina definisanja *let*-izraza, može se u potpunosti izbaciti upotreba tipa za identifikatore (`vname`) i koristi konstruktor sa narednim potpisom:

```
Let : exp -> (nat -> exp) -> exp
```

Kao prvi argument se očekuje validan izraz. Rezultat izračunavanja tog izraza se prosleđuje kao argument funkciji (`nat -> exp`). U pitanju je funkcija u jeziku *Gallina* koja ima ulogu tela naredbe `Let`. Induktivni princip za funkciju koja odgovara telu, daje naredni deo induktivne hipoteze:

```
forall (body : nat -> exp), (forall n : nat, P (body n))
```

Na taj način je demonstrirano da se mešovitim utapanjem mogu iskoristiti prednosti prethodna dva pristupa [2].

Zarad jasnoće daljih delova rada u kojima će često biti prikazivani blokovi koda kojima se praktično opisuju pojedini koncepti, u nastavku su data objašnjenja najosnovnijih konstrukcija programskog jezika u okviru dokazivača *Coq*.

2.5 Kratak pregled korišćenih konstrukcija *Coq*-a

U ovom poglavlju su navedeni osnovni pojmovi sistema *Coq* koji se koriste u nastavku rada. Jezik za specifikaciju *Gallina* je funkcionalni programski jezik koji se na osnovu *Kari-Hauardove* (engl. *Curry-Howard*) korespondencije[6] istovremeno može posmatrati i kao sistem u kojem je moguće izvođenje dokaza. *Coq* nudi moćan mehanizam za definisanje novih tipova. Svaki izraz je određenog tipa, a proveru je moguće sprovesti komandom `Check`.

Ključna reč `Inductive` uvodi novi tip. Konstruktori mogu posedovati parametre i time se omogućava stvaranje tipova koji su složeniji od enumerisanih konstanti. Prilikom kreiranja induktivnog tipa, *Coq* generiše induktivnu teoremu. Ugrubo, ukoliko se za proizvoljan predikat nad novim tipom dokaže da važi u svakom od *baznih* slučajeva (kada podatak odgovara jednom od konstruktora), tada se može zaključiti da predikat važi i kada je podatak bilo kog oblika koji odgovara definiciji tog tipa. Konstruktori sa više parametara, potencijalno rekurzivni, zahtevaju slojevitije objašnjenje. Prethodni mehanizam se naziva *induktivnim principom* i poglavlje *„Induction Principles"* u knjizi [18] daje precizniju formulaciju – postupno dajući složenije primere.

```
Notation "x ** y" := (pow x y) (at level 60, right associativity) : nat_scope.
```

Kôd 2.6: Primer uvođenja infiksnog zapisa operacije stepenovanja.

Komandom `Definition` je moguće dodeliti naziv nekom nerekurzivnom izračunavanju. Ključnom rečju `Notation` se uvode nove oznake za već postojeće definicije. Na primer, moguće je uvesti infiksnu notaciju operacije stepenovanja `pow`, kao u kodu 2.6. Za operaciju `pow` i za `x` i `y` tipa `nat` se omogućava infiksni zapis `x ** y`. Prioritet se definiše od 0 do 100 i manja vrednost označava viši prioritet, u ovom slučaju prioritet je 60. Operator može biti neasocijativan (`no associativity`), levo asocijativan (`left`) i kao što je u ovom primeru slučaj, desno asocijativan (`right`). Nekada je *Coq* u stanju da sâm zaključi koji su tipovi, ali mu je ponekad potrebno pomoći. Sa `nat_scope` nakon dvotačke je moguće naglasiti da je očekivani tip `nat`. Time se izbegavaju eventualni konflikti kada operatori iz istog imenskog prostora dele jedno ime. Dodatni primer korišćenja oznaka za tip je slučaj korišćenja niski,

[6] Detalji o logičkoj pozadini se mogu naći u radu [7].

za koje (zavisno od trenutnog imenskog prostora) može biti potrebno eksplicitno zadavanje načina tumačenja literala – nadovezivanjem sufiksa `%string`.

Navođenjem `Tactic Notation` moguće je uvesti novi zapis za taktike, odnosno kreirati taktiku višeg reda. Ovo može biti korisno ukoliko se uoči obrazac koji se ponavlja tokom dokaza.

Pisanje anonimnih funkcija se postiže sa rečju `fun`, nakon čega sledi lista parametara, a zatim simbol `=>` sa čije desne strane se nalazi povratna vrednost funkcije. Za rekurzivne funkcije je rezervisana ključna reč `Fixpoint`. U ovom slučaju treba napomenuti da nameću odgovarajuća sintaksička pravila, *svi* izrazi u jeziku *Gallina* bivaju zagarantovano zaustavljeni nakon izvršavanja. Pomenuto svojstvo *saglasnosti* (engl. *soundness*) je ključno kako bi sistem u kojem se dokazuje bio logički konzistentan.

```
Fail Fixpoint even (n : nat) : bool :=
       if 0 =? n then negb (even (n - 1)) else true.
(*  The command has indeed failed with message:
    Recursive definition of even is ill-formed ...
    Recursive call to even has principal argument equal to "n - 1"
        instead of a subterm of "n" ... *)
```

Kôd 2.7: Sintaksičko ograničenje za konstrukte `Fixpoint`.

Primer sintaksičkog zahteva koji mora biti ispunjen je da prilikom pisanja rekurzivnih funkcija pomoću konstrukta `Fixpoint`, barem jedan parametar ima zasigurno manju vrednost u odnosu na prethodni rekurzivni poziv. Sâm mehanizam kojim se proverava da li je taj zahtev ispunjen nije naročito napredan, pa se od korisnika očekuje da smanjivanje parametra bude što je eksplicitnije moguće. Na nužnost da to bude eksplicitno naznačeno ukazuje ispis greške iz koda prikazanog na 2.7. Iako u tom izrazu smanjivanje po `n` sa svakim novim rekurzivnim pozivom može delovati očigledno, *Gallina* nije u stanju da to zaključi. Izraz `(n-1)` po kojem se vrši naredni rekurzivni poziv ne raščlanjuje parametar `n` na jednostavniji oblik, već ga (sintaksički posmatrano) usložnjava. Prethodni izraz koristi infiksni operator oduzimanja, što je podrazumevano `Fixpoint sub` iz modula `Coq.Init.Nat`, pa ceo izraz *Gallina* interpretira kao `(sub n 1)`. Da bi se izraz uprošćavao u ovom smislu, neophodno je da se naredni rekurzivni poziv pravi za vrednost koja odgovara argumentima konstruktora iz induktivne definicije tipa. Direktnim raščlanjavanjem ulazne vrednosti,

korišćenjem konstrukcije `match n with` (umesto naredbe `if`), postiže se efekat sintaksičkog uprošćavanja.

Ovakav tip rekurzije se naziva *primitivnom rekurzijom* [19]. U slučaju znatno kompleksnijih izračunavanja koja sadrže i uzajamnu rekurziju, to može biti problem.

Pisanje interpretatorskih funkcija za neki jezik koji se modeluje, primer je u kojem prethodno ograničenje može predstavljati otežavajuću okolnost. Ako se interpretatorskoj funkciji doda parametar koji se zasigurno smanjuje, opisani problem biva prevaziđen. Takav pristup bi se mogao nazvati *interpretiranjem po koracima* (engl. *step-indexed evaluator*). Slikovito se upoređuje sa gorivom u rezervoaru – dozvoljava se interpretiranje izraza dokle god ima goriva, odnosno dok dodatni argument nije nula [18]. Ako u nekom trenutku izvršavanja argument postane nula, tada se funkcija zaustavlja sa porukom o grešci.

Za formulisanje tvrđenja koja se dokazuju mogu se koristiti ključne reči `Theorem` i `Lemma`. Dokaz takvih tvrđenja se izvodi nakon komande `Proof`, kojom se interaktivnim izvršavanjem ulazi u *režim dokazivanja* (engl. *proof mode*), iz kojeg se, po uspešno izvedenom dokazu, izlazi navođenjem komande `Qed`.

Ključnom rečju `Record` se opisuju slogovi. Standardnom bibliotekom *Coq*-a nije podržano ažuriranje vrednosti sloga, već je potrebno instalirati posebnu biblioteku. Korisnici alata za upravljanje paketima *opam* to mogu postići komandom: `opam install coq-record-update`.

Prethodno su spomenute neke od najosnovnijih konstrukcija jezika *Gallina*. *Coq* poseduje i druge delove, poput jezika za zadavanje taktika *Ltac*. Iskorišćavanjem mogućnosti prepoznavanja i upotrebe delova trenutnog *konteksta* (odnosno skupa hipoteza) i cilja, na sistematičan način se mogu zadavati strategije dokazivanja. To se postiže korišćenjem idioma `match goal with`, i analizom po slučajevima, nalik na uobičajen konstrukt za kontrolu toka `switch`. Pri korišćenju ovog dela *Coq*-a, redosled navođenja strategija može biti važan, jer zaustavljanje nije garantovano kao u jeziku *Gallina*.

U sklopu proširenja *Ltac* postoji veliki broj konstrukata, među kojima je i taktika višeg reda `repeat`. Kada je pozvana za neku (moguće višeg reda) taktiku `t`, u prvom koraku pokušava da tu taktiku primeni, a zatim da za sve dobijene potciljeve ponavlja postupak. Ukoliko u bilo kom koraku primena taktike `t` ne uspe ili bar ne promeni trenutni kontekst, postupak se završava.

Uputstva za korišćenje i informacije o mogućnostima dokazivača *Coq* se mogu naći u dokumentaciji [6]. Objašnjenja kroz konkretne primere data su u knjizi [5].

Glava 3

Konstrukcija jezika

U poglavlju koje sledi će kroz jednostavan primer biti praktično prikazano kako je formalno moguće definisati jezik. Cilj je da se davanjem isečaka koda u *Coq*-u (za pojedine aspekte od značaja) omogući formalno zaključivanje o samom jeziku koji se definiše. Za početak će induktivno biti zadata sintaksa jezika `Exp`.

3.1 Opšta postavka

Prilikom definisanja jezika, jedan od prvih koraka je zadavanje pravila koja određuju šta se smatra validnim, odnosno koji izrazi predstavljaju ispravne rečenice napisane u tom jeziku. Jedan način je navođenje kontekstno slobodne gramatike *BNF* (engl. *Backus–Naur form*) metaopisom pravila [19]. *BNF* predstavlja konvenciju za opisivanje ispravnih izraza, pritom ne odajući konkretne pojedinosti poput prioriteta ili asocijativnosti operatora istog prioriteta. Posmatranjem takvog opisa se brzo može steći uvid u njegovu apstraktnu sintaksu [18]. Zarad preciznosti treba napomenuti da postoje različite varijante *BNF*-a kojima je moguće zadati sadržajniji opis. Proširenom varijantom *EBNF* (engl. *Extended BNF*), između ostalog, dozvoljeno je korišćenje regularnih izraza, čime je zapis moguće unaprediti [19]. Ipak, posmatranjem samo *BNF*-a ostaju detalji koje je potrebno dodatno precizirati ukoliko je cilj otići korak dalje od apstraktne sintakse.

Implementirani ulazni jezik biće zasnovan na pojednostavljenju jezika *Prog* iz [14]. Na apstraktnom nivou, *Prog* je definisan narednom gramatikom:

$\langle exp \rangle$ ::= '`Const`' $\langle nat \rangle$
| '`Var`' $\langle vname \rangle$
| '`Op`' $\langle op \rangle$ [$\langle exp \rangle$]

| ‘`If`’ ⟨*test*⟩ [⟨*exp*⟩] ⟨*exp*⟩ ⟨*exp*⟩
| ‘`Let`’ ⟨*vname*⟩ ⟨*exp*⟩ ⟨*exp*⟩
| ‘`Call`’ ⟨*fname*⟩ [⟨*exp*⟩]

⟨*op*⟩ ::= ‘`Add`’ | ‘`Sub`’ | ‘`Mult`’ | ‘`Div`’ | ‘`Cons`’ | ‘`Head`’ | ‘`Tail`’

⟨*test*⟩ ::= ‘`Less`’ | ‘`Equal`’

Ovime je data pravilna forma ⟨*exp*⟩-rečenica koja se može direktno prevesti u duboko utapanje, u metajeziku nekog dokazivača. Smisao znakova ⟨⟩ i ‘’ iz gramatike je razlikovanje, redom, *pomoćnih* od *završnih simbola*.

Može se primetiti da je u ‘`Op`’, ‘`If`’ i ‘`Call`’ pravilu za ⟨*exp*⟩ korišćen i složen simbol [⟨*exp*⟩] koji ima značenje liste elemenata tipa ⟨*exp*⟩. Polimorfni tip liste standardne biblioteke *Coq*-a, instanciran baš za ⟨*exp*⟩ će u nastavku biti iskorišćen u tu svrhu. Na taj način će biti načinjeno blago odstupanje od dubokog utapanja u najstrožem smislu.

Simboli ⟨*nat*⟩, ⟨*vname*⟩ i ⟨*fname*⟩ su u [14] brojevi iz sintakse *Prog*. U tom smislu, simboli ⟨*vname*⟩ i ⟨*fname*⟩ odgovaraju brojčanim reprezentacijama, redom, naziva promenljivih i naziva funkcija iz konkretne sintakse. Istoimeni `nat` tip postoji i u sistemu *Coq*, a odgovara uobičajenom tipu prirodnih brojeva sa nulom.

Za potpun program i deklaraciju funkcije dodatno se uvode pravila:

⟨*prog*⟩ ::= ‘`Program`’ [⟨*dec*⟩] ⟨*exp*⟩

⟨*test*⟩ ::= ‘`Defun`’ ⟨*fname*⟩ [⟨*vname*⟩] ⟨*exp*⟩

U nastavku će najpre biti definisano značenje ⟨*exp*⟩-izraza.

Kako bi se uspostavilo formalno tumačenje prethodno definisane gramatike, potrebno je opredeliti se za okvir u kojem će validni izrazi biti tumačeni. Jedan pristup podrazumeva direktno prevođenje u neki *semantički domen* [12], sa oznakom Val, u kojem će sve vrednosti tipa ⟨*exp*⟩ biti tumačene. Praktično posmatrano, potrebno je definisati funkciju $eval : \langle exp \rangle \to Val$ u specifikacionom metajeziku dokazivača. To je način na koji se vrši *izračunavanje* ⟨*exp*⟩-izraza. Na primeru definisanja ‘`Const`’, ‘`Op`’ i ‘`If`’ biće prikazana moguća realizacija takvog tipa tumačenja u dokazivaču *Coq*.

3.2 Primer induktivno zadatog jezika Exp

Sama ⟨*exp*⟩-gramatika se može zadati induktivnim tipom `Exp`, kao u bloku koda 3.1. Vrednosni domen Val, u čijim će se vrednostima iskazivati tumačenja izraza u jeziku `Exp`, je takođe induktivan tip, nazvan `Val` i dat je kodom 3.2.

```
Inductive PrimOp : Type :=
| Add | Sub | Mult | Div | Cons | Head | Tail.

Inductive Compar : Type :=
| Less | Equal.

Inductive Exp : Type :=
| Const (n : nat)
| Op (op : PrimOp) (params : list Exp)
| If (test : Compar) (params : list Exp) (then_exp else_exp : Exp).
```

Kôd 3.1: ⟨*exp*⟩-gramatika zadata induktivnim tipom `Exp`.

```
Inductive Val : Type :=
| VNum (n : nat)
| VPair (v1 : Val) (v2 : Val).
```

Kôd 3.2: Induktivno zadat vrednosni domen Val.

Na ovakav način definisan tip `Val`, u osnovi oslikava strukturu binarnog stabla sa prirodnim brojevima u listovima, što predstavlja pogodnu reprezentaciju vrednosti za jezik nalik na Lisp. Svaka lista se može posmatrati kao uređeni par *glave* i *repa*. Time su operacije jezika apstrahovane unutrašnjim čvorovima stabla. U nastavku će biti prikazana dva načina tumačenja izraza u dokazivaču *Coq* i u tu svrhu biće korišćen polimorfni tip `option`. Njime se dobijaju dva oblika vrednosti, prvi oblik je `Some v`, gde je `v` onog tipa za koji je `option` instanciran, a drugi oblik je `None` – konstruktor bez parametara. Tipom `option Val` se postiže mogućnost razlikovanja dva tipa izračunavanja izraza u `Exp`-u. U slučajevima kada nema smisla izvesti neku operaciju biće vraćen `None`.

Interpretacija izraza u jeziku `Exp`

Nakon što su uvedeni tipovi za domen i kodomen funkcije *eval*, njoj samoj odgovara rekurzivno izračunavanje `eval` iz bloka koda 3.3. Iz potpisa funkcije `eval` se može uočiti da kodomen nije `Val` direktno, već je taj tip obmotan tipom `option`, zarad lakšeg detektovanja grešaka.

```
Fixpoint eval (exp : Exp) : option Val :=
  match exp with
  | Const n => Some (VNum n)
  | Op op [e] => eval_op op [eval e]
  | Op op [e1; e2] => eval_op op [eval e1; eval e2]
  | If test [e1; e2] e3 e4
    => match (eval e1), (eval e2) with
      | None, _ => None
      | _, None => None
      | Some (VNum n1), Some (VNum n2)
        => match test with
          | Less => if n1 <? n2 then (eval e3) else (eval e4)
          | Equal => if n1 =? n2 then (eval e3) else (eval e4)
          end
      | _, _ => None
      end
  | _ => None
  end.
```

Kôd 3.3: `Fixpoint` definicija funkcije *eval*.

Prethodnim tumačenjem sintaksički ispravnih izraza jezika `Exp` uvedena je takozvana *denotaciona semantika*. Denotaciona semantika definiše značenje prevođenjem u drugi jezik, za koji se pretpostavlja da je poznata semantika [12]. U ovom slučaju prevođenje se vrši u jezik *Gallina*. Funkcija `eval` koja preslikava izraze jezika `Exp` se naziva *funkcijom značenja* [12]. U prilogu 1 iz dodatka je navedena definicija za pomoćnu funkciju `eval_op`. Kada god je data neka vrednost u `Exp`-u, korišćenjem funkcije `eval`, lako je ustanoviti koja je odgovarajuća semantička vrednost. Razlog tome je što pomenuti tip semantike ima svojstvo *kompozitivnosti* – značenje jedne programske celine definisano je preko značenja njenih poddelova [12]. Trenutno definisana gramatika, u nedostatku konstrukata za promenljive još uvek ne poznaje pojam stanja, pa je njihovo pominjanje pri pravljenju ove semantike za sada izosta-

vljeno.

Još jedan tip semantike koji ima primenu u opisivanju programskih jezika je *prirodna operaciona* (engl. *big-step*) semantika. Naime, pomoću nje se opisuju ukupni rezultati izračunavanja [12]. Vodeći se tom idejom, u sistemu *Coq* se može opisati *svojstvo* zadato nad tipovima `Exp` i `Val` i takva relacija treba da važi onda kada se izraz tipa `Exp` izračunava u vrednost tipa `Val`. Bitna razlika u odnosu na prethodni (denotacioni) pristup je korišćenje relacija umesto funkcija. Blokom koda 3.4 je dat odsečak kojim se prikazuje jedan mogući način pravljenja te relacije, a u prilogu 2 se nalazi potpun kôd. Sa ovako zadatom relacijom, moguće je uveriti se u ispravnost tumačenja nekog konkretnog izvršavanja, takav primer je naveden u prilogu 3.

```
Reserved Notation "e '==>' v" (at level 90, left associativity).

Inductive evalExp_R : Exp -> Val -> Prop :=
| E_Const (n : nat) : (Const n) ==> (VNum n)

| E_OpAdd (params : list Exp) (e1 e2 : Exp) (n1 n2 : nat) :
  e1 ==> (VNum n1) ->
  e2 ==> (VNum n2) ->
  params = [e1; e2] ->
  Op Add params ==> VNum (n1 + n2)

| E_OpSub (params : list Exp) (e1 e2 : Exp) (n1 n2 : nat) :

(* ... nastavak u dodatku *)
```

Kôd 3.4: Isečak koda koji opisuje relaciju `evalExp_R`.

Pitanje koje se nameće je pitanje odabira – kako se opredeliti za okvir tumačenja? U slučajevima kada je izraze jednostavno opisivati pomoću funkcija, kao i u slučajevima kada se proceni da je isuviše komplikovano, odabir se sâm nameće [18]. Procenom da je potrebno podjednako truda, korišćenje induktivno definisanih relacija ima svoje prednosti – u slučaju da se koristi *Coq* [18].

Kao što je bilo pomena, svakom induktivnom tipu (pa i svojstvu) se kreiranjem pridružuje *induktivni princip*, što predstavlja moćan mehanizam za dokazivanje. Indukcija po izrazu u kom učestvuje relacija se može postići taktikom `induction`. Dodatno, kada se dokazuju svojstva jezika čija je semantika zadata relacijom, pred-

uslovi (iz relacije) bivaju automatski ubačeni u kontekst i to omogućava korišćenje taktike `inversion`. Njome se „razmotavaju" hipoteze induktivnog tipa iz konteksta. Automatski se dodaju jednakosti koje moraju da važe kako bi važila jednakost konstruktora za tip kojeg je hipoteza. Ako se pri tom uoče kontradikcije, dokazivanje trenutnog cilja se završava – pozivajući se na *princip eksplozije* (lat. *ex falso quodlibet*) [6, 18].

Jedna od prednosti denotacionog pristupa je u tome što su funkcije definisane za sve moguće ulaze, pa se to svojstvo ne mora posebno dokazivati. Dodatna prednost je mehanizam *ekstrakcije koda*[1] funkcija, ali i moguće sprovođenje lakših dokaza, koristeći mehanizme pojednostavljivanja računskih izraza. U većim projektima napisanim u *Coq*-u se upotrebljava više od jednog pristupa, a pomoćnim lemama se utvrđuje ekvivalentnost među njima, dok se za dokazivanje svojstava bira pogodniji oblik [18].

Magnus Mirin se u [14] opredelio za kompromis, gde umesto da svaku primitivnu operaciju posebno uvodi relacijom, koristi funkciju koja prima i `PrimOp` i objekat tipa `list Exp`. U nastavku će biti prikazan način proširenja jezika `Exp`, a u sledećoj glavi će biti definisan problem samokompilacije u IDT-u i način rešavanja.

3.3 Proširenje jezika `Exp`

Trenutno definisanom tipu `Exp` iz bloka koda 3.1 se jednostavno mogu pridružiti konstruktori `Var` i `Let`, kojima bi bilo podržano skladištenje, dohvatanje i ponovna dodela vrednosti promenljivama. Da bi se to postiglo, potrebno je uvesti pojam *okruženja* (ili *stanja*) programa, kojim bi se određivale valuacije promenljivih. U funkcionalnim jezicima se to može postići kreiranjem *parcijalnih mapa*. Parcijalne mape se kao u istoimenom poglavlju (engl. *„Maps: Total and Partial Maps"*) knjige [18] mogu definisati preko tipa *totalnih mapa*. Funkcije koje slikaju ključ u vrednost su dovoljne za opisivanje željene funkcionalnosti. Budući da je u ovom slučaju semantički domen fiksiran, umesto polimorfnog tipa mape, moguće je napraviti konkretan tip `Env` kao u kodu 3.5.

Okruženje je, dakle, funkcija kojom se identifikator promenljive (tipa `string`) preslikava u vrednost te promenljive (tipa `option Val`). Koristi se `option`, da se naglasi slučaj kada promenljiva nije definisana i tada se vraća `None`. Prazno okru-

[1] *Coq* poseduje mogućnost prevođenja koda funkcija – u jezike *Haskell*, *OCaml* ili *Scheme*, čime se, na primer, može dobiti izvršivi oblik semantičkih funkcija [6].

ženje `empty_env` opisuje situaciju u kojoj nijedna promenljiva nije inicijalizovana. Stoga se ono definiše kao konstantna funkcija koja uvek vraća `None`. Da bi se u tako uspostavljenom okruženju mogle vršiti dodele vrednosti, potrebno je da se izgrađuje funkcija koja slika ključ u vrednost, što je omogućeno sa `update_env`. Treba primetiti da se primenom funkcije automatski dobija operacija dohvatanja vrednosti za neku promenljivu [18]. Komandom `Notation` se dobija prigodna notacija za vršenje prve inicijalizacije i vršenja dodela vrednosti inače. U prilogu 4 na strani 59 je dat primer korišćenja okruženja.

```
Definition Env := string -> option Val.
Definition empty_env : Env := (fun _ => None).
Definition update_env (e : Env) (id_new : string) (v : option Val) :=
  fun id => if eqb_string id id_new then v else (e id).

Notation "id '|->' val" := (update_env empty_env id val) (at level 100).
Notation "id '!->' val ';' env" := (update_env env id val)
    (at level 100, val at next level, right associativity).
```

Kôd 3.5: Struktura kojom se čuvaju vrednosti promenljivih.

Ovako kreiranom podrškom za promenljive je nadalje moguće proširiti postojeću implementaciju dodavanjem narednih elemenata:

- Novih konstruktora za `Exp`:
  - `Var (v_id : vname)`
  - `Let (v_id : vname) (assign_exp body_exp : exp)`
- Okruženja funkciji `eval`, za koje se izraz izračunava
- Okruženja levoj strani relacije `evalExp_R`
- Parametra tipa `Env` pomoćnim lemama iz naredne glave

Iskorišćavanjem ovakve infrastrukture, podrška za *let*-izraz se sprovodi na sličan način kao i kod prethodnih izraza.

U nastavku je predstavljen način generisanja koda, kao i definisanja i dokazivanja tvrdnji o ispravnosti prevođenja samokompilatora datog u vidu funkcije u jeziku IDT-a. Pristup iz narednog poglavlja u velikoj meri prevazilazi praktične okvire ovog master rada, ali su detaljno analizirani korišćeni koncepti i neki praktični elementi.

Glava 4

Postavka i rešenje problema samokompilacije

Celokupan način definisanja problema preuzet je iz rada o verifikovanoj samokompilaciji [14]. Korišćenje interaktivnih dokazivača i drugačijih tehnika za postizanje automatskog prevođenja programskog koda između izraza datih u različitim tipovima utapanja, postoji i u drugim radovima [8, 4]. Originalna ideja za *ovakav* način prevođenja data je u zasebnom radu iz 2012. godine [16].

Prevodilac za ulazni jezik $Prog$, napisan u okvirima metajezika, predstavljen je funkcijom koja za datu nisku (konkretne sintakse $Prog$) na ulazu vraća takođe nisku, ali u konkretnoj sintaksi ciljnog jezika $Targ$. O takvoj funkciji je prirodno razmišljati kao o kompoziciji potfunkcija, gde svaka pojedinačna odgovara nekoj od faza prevođenja:

$$compiler(s) = (targ2str \circ codegen \circ parser \circ lexer)(s)$$

U slučaju rudimentarnog ulaznog jezika koji podseća na Lisp, funkcija koja ima ulogu leksera, zajedno sa funkcijom koja parsira te tokene, može biti objedinjena u jednu. Funkcija koja prevodi apstraktnu sintaksu ulaznog jezika, u apstraktnu sintaksu ciljnog jezika, zapravo vrši prevođenje iz jednog *dubokog utapanja* u drugo. Na kraju, potrebna je i $targ2str$ funkcija koja će konkretizovati sintaksu ciljnog jezika, rezultujući niskom koja daje izvršivi kôd. Sâm proces je u izvesnom smislu inverzan transformacijama koje se odvijaju pre funkcije *codegen*. Umesto od konkretne ka apstraktnoj, prevođenje je u suprotnom smeru (i različiti su jezici). U [14] je za ciljni jezik odabran podskup asemblera *x86-64*.

Notacija koja se upotrebljava za iskazivanje tvrđenja o radu programa se oslanja

na semantike $\Downarrow_{Prog}$ i $\Downarrow_{Targ}$ koje su definisane u nastavku. Zapis

$$\vdash (input, p) \Downarrow_{Prog} output$$

podrazumeva da se program p sa zadatim ulaznim podacima *input* na standardnom ulazu *izvršava i zaustavlja* i daje izlaz *output* na standardnom izlazu. Primer programa koji ispisuje jedno slovo: $\vdash$ (””, ‘`Program`’[](‘`Op`’‘`Write`’[‘`Const`’97])) $\Downarrow_{Prog}$ ”a”.

Na isti način se tumače i tvrđenja sa relacijom $\Downarrow_{Targ}$, o programima u apstraktnoj sintaksi $Targ$. Ciljni jezik i relacija $\Downarrow_{Targ}$ će biti navedeni samo u opštim terminima. Razlog tome je što izuzev *oblika*, koji je istovetan relaciji $\Downarrow_{Prog}$, dodatni opisi nemaju uticaja na dalji tok teksta ovog poglavlja.

Za izraze tipa Exp relacija je nešto složenijeg oblika:

$$\vdash (env, exps, s_{start}) \Downarrow_{Exp} (vals, s_{end})$$

Pojašnjenje značenja relacije $\Downarrow_{Exp}$ zahteva prethodno opisivanje elemenata koji figuriraju u relaciji. Izraz *exps* predstavlja listu izraza tipa Exp, a $vals$ listu izraza tipa Val. Na ovaj način se povezuje više izraza sa njihovim rezultatima. Dato okruženje *env* se može predstaviti na prethodno opisan način (kao u kodu 3.5). Okruženjem su, dakle, definisane vrednosti promenljivih. Na primer, $env\ n = Some\ v$ označava da je vrednost promenljive n jednaka v.

Sa s je obeleženo trenutno *stanje programa* i ono se može predstaviti slogom koji sadrži po jednu nisku za opis ulaznih i izlaznih vrednosti, kao i listu deklaracija funkcija. Odgovarajući tip bi mogao da bude nalik na:

$$\begin{aligned} state = <|\ & input : string; \\ & output : string; \\ & decls : list\ decl\ |> \end{aligned}$$

Polja *input* i *output* odgovaraju standardnom ulazu i izlazu i tipa su niske ($string$), dok polje *decls* predstavlja listu deklaracija i upravo tog je tipa ($list\ decl$). Na primer, sa

$$\begin{aligned} s_x &\stackrel{\text{def}}{=\!=} <|input := "hello"; output := ""; funs := [print\_decl]; |> \\ s_y &\stackrel{\text{def}}{=\!=} <|input := "ello"; output := "h"; funs := [print\_decl]; |> \end{aligned}$$

su opisana neka dva stanja s_x i s_y. Oba stanja poseduju deklaraciju funkcije za štampanje $print\_decl$[1], a na ulazu imaju redom niske „hello” i „ello”. Na standardni izlaz s_x nije ništa ispisano, a na izlazu od s_y je ispisano slovo „h”.

[1]Konkretna definicija je data u kodu 5.4 u poglavlju o realizaciji 5.

Znajući značenje prethodno navedenih elemenata, konačno je moguće protumačiti i sveukupni izraz kojim se uspostavlja relacija između objekata sa leve i objekata sa desne strane znaka $\Downarrow_{Exp}$. Počevši od stanja s_{start} i okruženja env, lista izraza tipa Exp, odnosno $exps$ se izvršava i zaustavlja u nekom završnom stanju s_{end} izračunavajući se u vrednosti $vals$. Primer izraza kojim se na standardni izlaz ispisuje jedno slovo sa standardnog ulaza: $\vdash (env, [\text{`}\texttt{Op}\text{'`}\texttt{Write}\text{'}[\text{`}\texttt{Op}\text{'`}\texttt{Read}\text{'}[]]], s_x) \Downarrow_{Exp} ([\text{VNum } 0], s_y)$.

4.1 Opšti okvir za prevođenje sa jezika IDT

Funkcija se naziva *čistom* kada nema bočnih efekata – što u ovom slučaju podrazumeva funkcije koje ne vrše promene nad strukturom stanja s. Pritom jedine promene koje se mogu načiniti nad tom strukturom su izmene niski u kojima se skladište ulaz i izlaz. Prilikom razmatranja čistih funkcija u ovom kontekstu, ključni princip koji se najviše eksploatiše u [14] i zauzima centralno mesto u [16], zahteva veću pažnju pre detalja o tvrđenjima korektnosti faza prevođenja.

Za dat izraz na jeziku dokazivača, potrebno je pronaći odgovarajući izraz u apstraktnoj sintaksi jezika $Prog$. Krajnji cilj je tako prevesti funkciju *compiler*.

Ukratko, potrebno je snabdeti dokazivač lemama specijalnog oblika, na osnovu kojih bi pažljivo napravljenom automatizacijom postupka dokazivanja (i bez dodatnog zalaženja u semantiku), dokazivač bio taj koji pronalazi prevod iz jedne u drugu reprezentaciju. Krajnje teoreme koje se takvim postupkom dokazivanja dobijaju predstavljaju vezu dveju reprezentacija koda i nazivaju se *sertifikatima*. To je potvrda da dobijena reprezentacija odgovara izrazu koji se prevodi. Konkretnije, specijalne leme koje je potrebno samostalno dokazati i uvesti u bazu „činjenica" su sledećeg oblika:

$$\vdash (env, [x], s) \Downarrow_{Exp} ([encoding\ tm], s) \tag{4.1}$$

Primećuje se da stanje s ostaje nepromenjeno. Uobičajena notacija uglastih zagrada za predstavljanje lista je umetnuta kako bi se zadovoljila forma relacije $\Downarrow_{Exp}$. Jedini do sada nerazjašnjen deo tvrđenja je $encoding\ tm$, sa desne strane relacije. U pitanju je funkcija *encoding* kojom se enkodira izraz tm zapisan na metajeziku, tj. izračunava se njegova semantička vrednost tipa Val.

Najjednostavniji primer funkcije enkodiranja je onaj kojim se vrši neposredno preslikavanje „podudarnih" tipova. Za tipove koji postoje i u Val i u metajezičkom

sistemu, moguće je iskoristiti konstruktor tipa Val, čime se može dobiti željeno tvrđenje.

Primera radi, iz definicije gramatike s početka rada, prvi i najjednostavniji slučaj validnog $\langle exp \rangle$-izraza je dat u prvom redu zapisa u *BNF*-u sa: $\langle exp \rangle ::= \text{`Const'} \langle nat \rangle$. Izrazi tipa istinitosne vrednosti se *Bool* funkcijom mogu prevesti u one vrednosti iz $Val$ koje odgovaraju brojevima 1 i 0 (kao u kodu 4.1). Na taj način se dobija naredno tvrđenje koje je moguće dokazati: $\vdash (env, [\text{`Const'}1], s) \Downarrow_{Exp} ([Bool\ \top], s)$.

4.2 Prevođenje od *Gallina* ka jeziku `Exp`

Nakratko će opšti okvir predstavljanja tehnike biti zamenjen primerima realizacije prethodno date postavke opisane u jeziku *Gallina*, u terminima konstrukata koji se tiču tipa `Exp` i odgovarajuće relacije `evalExp_R`. Ubrzo, kako se primeri budu usložnjavali, tako će opšti okvir (nezavisan od metajezika) biti jednostavniji za tumačenje i njime će tehnika biti objašnjavana.

Odgovarajuća sintaksa i interpretacija u metajeziku *Gallina* je data kodom 3.1 i 3.4. Da bi se izraz koji sadrži samo konstantu tipa `nat`, preveo u semantički domen `Val` može se iskoristiti prvi konstruktor relacije `evalExp_R`, na strani 21. Konkretno, automatizaciji je dovoljan naredni konstruktor da prevede izraze koji su tipa numeričkih literala:

```
E_Const (n : nat) : (Const n) ==> (VNum n)
```

Numerički literali su specijalan slučaj jer je lako uvideti da su konstruktor `Val`-a `VNum (n : nat)` i konstruktor `Exp`-a `E_Const (n : nat)` isti. Kada tip iz metajezika ne postoji, kao što je slučaj sa tipom `bool`, tada je potrebno napraviti funkciju enkodiranja poput one date u kodu 4.1. Njome bi se literali `true` i `false` lako mogli prevesti. Oba pomenuta slučaja su najjednostavnija jer će u bilo kojem izrazu biti na samom dnu *apstraktnog sintaksičkog stabla* (engl. *Abstract Syntax Tree*, u nastavku *AST*) kao listovi.

```
Definition Bool (b : bool) : Val :=
  match b with | true => VNum 1 | false => VNum 0 end.
```

Kôd 4.1: Funkcija enkodiranja izraza tipa `bool`.

Prvi sledeći izrazi prema složenosti su oni koji se dobijaju direktnom primenom neke operacije nad listovima. Slučaj unutrašnjih čvorova je složeniji, ali ako se posmatraju primitivne operacije za koje takođe postoji odgovarajući konstruktor relacije `evalExp_R`, obrazac prevođenja ostaje isti. Nešto zanimljiviji slučaj je kada se jednostavna operacija izvršava nad tipom za koji je neophodna funkcija prevođenja. Na primer, posmatrajmo izraz tipa `bool` i jednostavnu operaciju negiranja `negb` koju pruža *Gallina*. U tom slučaju je potrebno navesti lemu 4.2.

```
Lemma negating_bool_lemma : forall (b : bool) (e : Exp), e ==> (Bool b) ->
  (Op Sub [Const 1; e]) ==> (Bool (negb b)).
Proof.
  intros; destruct b; simpl in *; repeat econstructor; eauto.
Qed.
```

Kôd 4.2: Lema o negiranju izraza tipa `bool`.

Deskriptivniji dokaz leme, za na ovaj način pretpostavljene tipove, dat je u prilogu 5. Uz poštovanje preduslova `e ==> (Bool b)`, izraz `Op Sub`, sa leve strane relacije `==>`, opravdava značenje operacije `negb` primenjene na bulovski izraz `b`.

```
Lemma if_equal_lemma_n : forall (params : list Exp) (e1 e2 : Exp)
                     (n1 n2 n3 n4 : nat) (then_exp else_exp : Exp),
 params = [e1; e2] ->
 e1 ==> (VNum n1) -> e2 ==> (VNum n2) ->
 then_exp ==> (VNum n3) -> else_exp ==> (VNum n4) ->

 (If Equal params then_exp else_exp) ==> VNum (if n1 =? n2 then n3 else n4).
```

Kôd 4.3: Lema za slučaj `If Equal`, kad se u obe grane dobija broj.

Najzad, na istovetan način se može razmotriti i poslednji slučaj pravila induktivno definisanog tipa relacije iz koda 3.1. Najpre je potrebno dodati leme koje se tiču slučaja `If`. Tip rezultata `if-then-else` naredbe nije uvek isti, ali recimo da se razmatraju samo one situacije kada obe grane rezultuju brojčanom vrednošću i kada je uslov poređenja iskazan operatorom `=?`. Izraz tipa `Exp` koji to oslikava imaće argument `(test : Compar)` s vrednošću `Equal`. Odgovarajuća lema se može videti u kodu 4.3, a dokaz je dat u prilogu 6.

Na isti takav način se obrađuje i slučaj kada se grane izračunavaju u bulovsku vrednost. Naime, tipovi u potpisu leme bi se neznatno razlikovali, `n3` i `n4` bi bili tipa `bool`, ceo izraz bi bio pod funkcijom enkodiranja `Bool`, a dokaz je gotovo istovetan. Iz tog razloga je za navedeno tvrđenje iz priloga 7 izostavljen dokaz. Analogno se sprovodi i uvođenje preostalih lema koje se odnose na slučaj `If Less`.

Uz sve do sad navedene pomoćne leme moguće je napisati automatizovan postupak dokazivanja, kojim bi se dobijao prevod. To se postiže korišćenjem dodatnog sloja apstrakcije podržanim od strane dokazivača *Coq*. Sa prethodno navedenim lemama i dobro odabranim redosledom primena taktika, sistem bi, bez dodatne ljudske pomoći bio u stanju da izvrši prevod u sintaksu zadatu tipom `Exp` iz bloka koda 3.1. Kompletna automatizacija za dosadašnji jezik data je u bloku koda 4.4.

```
Ltac use_lemmas_auto :=
  match goal with
  | [ |- ?E ==> VNum (?M + ?N) ] => eapply E_OpAdd
  | [ |- ?E ==> VNum (?M - ?N) ] => eapply E_OpSub
  | [ |- ?E ==> VNum (mult ?M ?N) ] => eapply E_OpMult
  | [ |- ?E ==> VNum (div ?M ?N) ] => eapply E_OpDiv

  | [ |- context[?X ==> VNum (if ?Y =? ?Z then ?V else ?W)] ] =>
      eapply if_equal_lemma_n
  | [ |- context[?X ==> Bool (if ?Y =? ?Z then ?V else ?W)] ] =>
      eapply if_equal_lemma_b
  | [ |- context[?X ==> VNum (if ?Y <? ?Z then ?V else ?W)] ] =>
      eapply if_less_lemma_n
  | [ |- context[?X ==> Bool (if ?Y <? ?Z then ?V else ?W)] ] =>
      eapply if_less_lemma_b

  | [ |- context[?X ==> Bool true] ] => eapply bool_true_lemma
  | [ |- context[?X ==> Bool false] ] => eapply bool_false_lemma
  | [ |- context[?X ==> Bool (negb ?B)] ] => eapply negating_bool_lemma

  | [ |- ?E ==> ?V ] => econstructor; eauto
  | _ => eexists; intros
  end.
```

Kôd 4.4: Automatizacija postupka prevođenja na jezik opisan kodom 3.1 i 3.4.

Blokom koda 4.5 se prikazuje način na koji je moguće iskoristiti prethodnu stra-

tegiju za prevođenje.

```
Lemma simple_if_expression : exists (x : Exp),
  x ==> Bool (if (5 <? (3 + 2 - 1))
              then (negb true) else (negb (negb false))).
Proof. repeat use_lemmas_auto. Show Proof. Qed.
```

Kôd 4.5: Prevođenje konkretnog izraza sa naredbom `if`.

Već pre komande `Show Proof` tvrđenje biva dokazano, a njenim izvršavanjem se na ekranu prikazuje svedok tačnosti. U pomenutom slučaju to je poduži izraz iz koda 4.6.

```
(If Less [Const 5; Op Sub [Op Add [Const 3; Const 2]; Const 1]]
  (Op Sub [Const 1; Const 1]) (Op Sub [Const 1; Op Sub [Const 1; Const 0]]))
```

Kôd 4.6: Svedok tačnosti tvrđenja iz koda 4.5.

4.3 Rekurzivni slučaj prevođenja

U nastavku će ponovo biti korišćen uopšteniji tok, nezavisan od odabira metajezika. Već dokazano tvrđenje `negating_bool_lemma` ima sledeći oblik:

$$\vdash (env, [x], s) \Downarrow_{Exp} ([Bool\ b], s) \Rightarrow (env, [\text{`Op'}\ \text{`Sub'}\ [\text{`Const'}1; x]], s) \Downarrow_{Exp} ([Bool(\neg b)], s)$$

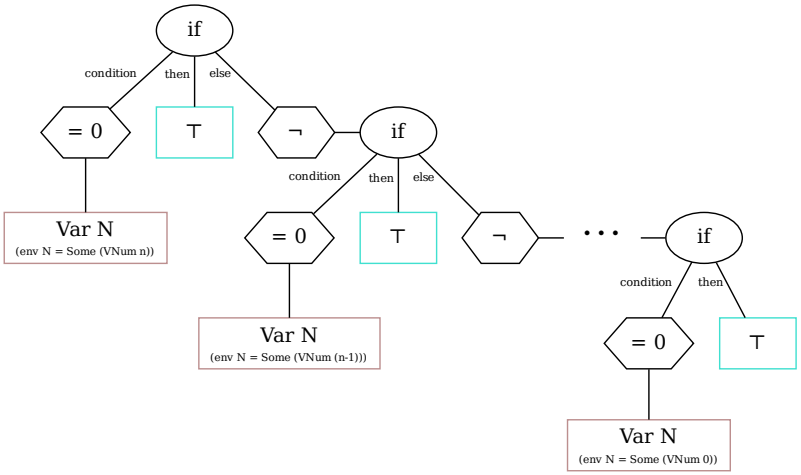
Svi primeri lema koji su dodavani kako bi se automatizacija mogla izboriti sa dokazom se uklapaju u opštu formulaciju (4.1). Jedina razlika je to što su okruženje i stanje, kao nepodržani konstrukti bili ignorisani, a relacija se nije odnosila na listu, već na pojedinačan izraz.

Ovakav način dobijanja prevoda koda je razjašnjen za konstrukte koji ne podrazumevaju rekurziju, ali šta se događa u slučajevima kada postoje rekurzivni pozivi? Tehnika prevođenja kojom je i to obuhvaćeno će kao i u radovima [16] i [14] biti razložena na prostom primeru rekurzivne funkcije.

Neka je narednom definicijom data funkcija *even* nad prirodnim brojem n, koja vraća istinitosnu vrednost $\top$ ili $\bot$, u zavisnosti od toga da li je n paran ili nije.

$$even\ n \overset{\text{def}}{=\!=} if\ (n\ =\ 0)\ then\ \top\ else\ \neg even\ (n-1) \tag{4.2}$$

Da bi automatsko prevođenje ovog izraza bilo podržano, potrebna je funkcija enkodiranja bulovskog tipa, leme za generisanje izraza koji odgovaraju bulovskim konstantama, lema za negaciju bulovskog izraza, lema za uslov grananja, a u ovom slučaju je potrebna i izvesna podrška za prevođenje aritmetičkih izraza i promenljivih. Aritmetički izrazi isključivo sa konstantama (ne i promenljivama) mogu biti prevedeni ili direktnim preslikavanjem (konstante metajezika u konstantu jezika koji se modelira) ili prevođenjem rezultata izraza, koji je takođe konstanta, u konstantu jezika koji se modelira. Ukoliko se u aritmetičkim izrazima pojavljuju i promenljive, njih je potrebno uočiti i enkodirati. Jedan pristup podrazumeva određeni uvid u kôd funkcije koja se prevodi. Oblik kvantifikovanog izraza se u tom slučaju može prilagoditi tome da u izrazu metajezika postoje promenljive na određenim mestima. Takav pristup će biti korišćen prilikom demonstracije prevođenja u poglavlju o realizaciji. U suprotnom, ukoliko je moguće razmatrati *AST* izraza metajezika, onda se polazeći od listova uočavaju isključivo konstante i promenljive. Na slici 4.7 je uz primer apstraktnog izvršavanja funkcije *even*, u listovima prikazana i vrednost promenljivih[2].



Slika 4.7: *AST* rekurzivne funkcije `even`.

Naziv promenljive (u obliku niske) se može enkodirati brojem. Na primer, ukoliko su dozvoljeni karakteri iz osnovne *ASCII* tabele sa 128 vrednosti, tada se svaka niska može zamisliti kao broj zapisan u osnovi 128, gde su brojčane vrednosti karaktera vrednosti cifara. Vrednost takvog broja jednoznačno određuje svaku nisku i obrnuto.

[2]Zarad doslednosti, prikaz vrednosti promenljive odgovara jeziku koji se modelira i sadrži oznaku lokalnog okruženja `env`.

Kada se u listu *AST*-a uoči promenljiva, tada se ona trivijalno prevodi (posebnom lemom) u odgovarajuću promenljivu jezika koji se modeluje.

Nakon što je obrađen slučaj promenljivih, aritmetičke operacije nad (već prevedenim) konstantama i promenljivama se obrađuju isto kao i operacija negiranja bulovskog izraza – uvođenjem lema. Na slici 4.7 prosleđivanje argumenta nije eksplicitno prikazano, već je ažurirana lokalna vrednost promenljive.

Kada je stvorena baza takvih lema, postupak prevođenja funkcije *even* započinje dokazivanjem narednog tvrđenja:

$$\begin{aligned} &\vdash env\ N = Some\ (\text{VNum}\ n) \wedge \\ &\quad (n \neq 0 \Rightarrow app\ \text{EVEN}\ [\text{VNum}\ (n-1)]\ s\ (Bool\ (even\ (n-1)),\ s)) \Rightarrow \\ &\quad (env,\ [even\_body],\ s) \Downarrow_{Exp} ([Bool\ (\textbf{even n})],\ s) \end{aligned} \tag{4.3}$$

Prvi red predstavlja pretpostavku tvrđenja da je promenljivoj sa nazivom „n", koja je enkodirana brojčanom vrednošću N, dodeljena neka konkretna vrednost n, što je u vrednosnom domenu Val izraženo kao (VNum n). Drugi red sadrži pretpostavku u formi implikacije. Simbol EVEN je oznaka enkodirane vrednosti naziva funkcije *even* iz metajezika. Međutim, pošto takav simbol nema dodeljeno nikakvo drugo značenje vezano za *ponašanje* funkcije *even*, neophodno je uvesti pretpostavku da se (za proizvoljne argumente) primenom funkcije EVEN dobija isti rezultat kao i nakon primene *even*. U jeziku koji se modelira, EVEN označava identifikator funkcije koja odgovara funkciji *even*. Relacija *app* dovodi u vezu, narednim redosledom pobrojane elemente:

- Identifikator funkcije: EVEN
- Listu vrednosti argumenata na koje se funkcija primenjuje: [VNum $(n-1)$]
- Zadato početno stanje: s
- Uređen par koji opisuje rezultat i efekte primene funkcije
  - Vrednost rezultata primene funkcije: $Bool\ (even\ n)$
  - Stanje nakon primene funkcije[3]: s

Implikacijom iz drugog reda se tvrdi da ukoliko važi da je konkretna brojčana vrednost n različita od nule, onda važi i upravo opisana desna strana te implikacije. Ova

[3]U slučaju čistih funkcija poput *even*, početno stanje ostaje nepromenjeno.

pretpostavka odgovara rekurzivnom slučaju funkcije *even*. Razlog definisanja nove relacije *app*, proishodi iz potrebe da se o nepoznatoj funkciji (koja će tek naknadno biti generisana), zaključuje pomoću *vrednosti prosleđenih argumenata*, a ne na osnovu *identifikatora parametara* te funkcije[4].

Umesto celog tela funkcije *even n* (desne strane iz definicije (4.2)), zarad preglednosti je navedena oznaka **even n**. Sa *even_body* je označen izraz koji je potrebno dobiti. Ako su oba konjunkta pretpostavke zadovoljena, potrebno je dokazati da tada važi i relacija $\Downarrow_{Exp}$ iz trećeg reda, što je uz dovoljan broj odgovarajućih lema moguće prepustiti automatizovanom procesu. Tako se dobija telo funkcije, za koju znamo da odgovara identifikatoru EVEN. Uzevši u obzir način definisanja stanja (dat na strani 25) kao sloga koji sadrži ulaz, izlaz i deklaracije funkcija, teorema (4.3) se može preformulisati. Deklaracija funkcije povezuje vrednosti identifikatora parametara sa konkretnim telom funkcije. U telu funkcije mogu biti prisutne promenljive sa tim identifikatorima. Polazeći od već dokazane teoreme (4.3), posedovanjem konkretnog *even_body* i eliminisanjem ručno dodatih pretpostavki o vezi EVEN i *even*, moguće je dokazati narednu teoremu:

$$\begin{aligned} &\vdash lookup\_fun\ \text{EVEN}\ s.decls\ = Some\ ([N], even\_body) \wedge \\ &\quad (n\ \neq\ 0\ \Rightarrow\ app\ \text{EVEN}\ [\text{VNum}\ (n-1)]\ s\ (Bool\ (even\ (n-1)),\ s)) \Rightarrow \\ &\quad app\ \text{EVEN}\ [\text{VNum}\ (n)]\ s\ (Bool\ (even\ n),\ s) \end{aligned} \tag{4.4}$$

Deo pretpostavke iz prvog reda sadrži novu oznaku $lookup\_fun$ koja predstavlja funkciju iz metajezika. Njena uloga je da za prosleđen identifikator funkcije i listu sa deklaracijama funkcija, u toj listi pronađe onu deklaraciju koja odgovara identifikatoru. Rezultat koji se vraća je obmotan tip, zbog mogućnosti razlikovanja slučaja kada deklaracija sa traženim identifikatorom nije sadržana u prosleđenoj listi. Ono što se u slučaju uspešnog pronalaženja vraća je potpun opis deklaracije – lista enkodiranih vrednosti identifikatora parametara i izraz koji odgovara telu funkcije. Konačno, pozivajući se na dokaz o zaustavljanju *even* funkcije, moguće je iskoristiti induktivni princip i osloboditi se dela pretpostavke iz drugog reda teoreme (4.5), dobivši na kraju:

$$\begin{aligned} &\vdash lookup\_fun\ \text{EVEN}\ s.decls\ = Some\ ([N], even\_body) \Rightarrow \\ &\quad app\ \text{EVEN}\ [\text{VNum}\ (n)]\ s\ (Bool\ (even\ n),\ s) \end{aligned} \tag{4.5}$$

[4]Kroz ceo rad se podrazumeva, ali tek nadalje postaje bitno razlikovati pojmove *argumenata* i *parametara*, kao prosleđenih vrednosti nekoj funkciji i identifikatora iz definicije (tela) funkcije.

Upravo opisani postupak prevođenja iz jednog u drugi jezik, moguć je i u suprotnom smeru. Primer *even* funkcije je direktno preuzet iz rada [14], a prevođenje iz dubokog utapanja u validne izraze metajezika, je uz pomoć takođe jednostavne funkcije *append*, dato u radu [16]. U poglavlju o realizaciji će biti približeno kako je u *Coq*-u moguće sprovesti sličnu strategiju.

4.4 Formulisanje tvrđenja korektnosti

Vraćajući se na definiciju funkcije parsiranja, ispravnost svih pomenutih potfunkcija osim *codegen* je garantovana njihovim (konceptualno istim) načinom dobijanja. Funkcija *codegen* zahteva posebno razmatranje, stoga se navodi tvrđenje *o ispravnosti generatora koda* koje će kasnije biti dokazano:

$$\begin{aligned} &\vdash (input, p) \Downarrow_{Prog} output_1 \wedge \\ &(input, codegen\ p) \Downarrow_{Targ} output_2 \\ &\qquad \Rightarrow output_1 = output_2 \end{aligned} \tag{4.6}$$

Ova teorema daje vezu između izvršivih programa p u sintaksama $Targ$ i $Prog$ i dovoljna je za glavno tvrđenje o tačnosti celokupnog prevodioca. Može se primetiti da se na ovaj način ne daju garancije da će ako se program p izvršava u sintaksi $Prog$, semantikom $\Downarrow_{Prog}$, to biti slučaj i za *codegen* p u semantici $\Downarrow_{Targ}$. Za takve jače garancije potrebno je posebno obraditi programe koji se ne izvršavaju. To se u radu [14] postiže uvođenjem *semantika divergencije* $\Uparrow_{Prog}$ i $\Uparrow_{Targ}$ i odgovarajućih tvrđenja sa njima – u ovom master radu to neće biti razmatrano.

Nadalje je potrebno rešiti problem samokompilacije. Na osnovu trenutno uvedenih pojmova, a usled nepoklapanja tipova[5] – to nije moguće. Zato se uvodi nova konstanta $compiler_{Prog}$. Njen smisao je da predstavlja prevedenu funkciju *compiler* u jezik apstraktne sintakse $Prog$ i zato je potrebno da bude definisana tako da naredno tvrđenje *o korektnosti konstante* $compiler_{Prog}$ bude zadovoljeno:

$$\vdash (input,\ compiler_{Prog}) \Downarrow_{Prog} (compiler\ input) \tag{4.7}$$

Sledeći korak je dobijanje izvršivog koda, definisanjem konstante $compiler_{Targ}$, odnosno predstavljanjem funkcije *compiler* u apstraktnoj sintaksi ciljnog jezika

[5]Pokušaj samokompilacije bi bio primena funkcije *codegen* na *compiler*, u želji da se dobije izvršivi kôd. Međutim, umesto apstraktne sintakse *Prog* koju očekuje, funkciji *codegen* bi, takođe, bila prosleđena funkcija (koja jednu nisku preslikava u drugu).

$Targ$. Na osnovu već pretpostavljenih elemenata definiše se kao:

$$compiler_{Targ} \stackrel{\text{def}}{=\!=} (codegen\ compiler_{Prog}) \tag{4.8}$$

Sada je samokompilacija zaista moguća, iskorišćavanjem tvrđenja (4.6) i (4.7). Unifikovanjem slobodne promenljive p sa konstantom $compiler_{Prog}$ dobija se:

$$\begin{aligned} \vdash (input, compiler_{Prog}) \Downarrow_{Prog} (compiler\ input) \wedge \\ (input, codegen\ compiler_{Prog}) \Downarrow_{Targ} output_2 \\ \Rightarrow (compiler\ input) = output_2 \end{aligned} \tag{4.9}$$

Ako se dodatno iskoristi definicija (4.8) za konstantu $compiler_{Targ}$ i ako se proizvoljno odabrana oznaka $output_2$ za izlaz promeni u $output$, (4.9) poprima lepšu formu:

$$\begin{aligned} \vdash (input, compiler_{Prog}) \Downarrow_{Prog} (compiler\ input) \wedge \\ (input, compiler_{Targ}) \Downarrow_{Targ} output \\ \Rightarrow (compiler\ input) = output \end{aligned} \tag{4.10}$$

Pritom, prvi konjunkt sa leve strane implikacije je uvek tačan, zbog tvrđenja o korektnosti $compiler_{prog}$ (4.7), pa je krajnji oblik *teoreme o samokompilaciji*:

$$\vdash (input, compiler_{Targ}) \Downarrow_{Targ} output \ \Rightarrow\ compiler\ input = output \tag{4.11}$$

4.5 Postupak izvršavanja koda

Da bi kôd kompilatora mogao da se pokrene, konstanta $compiler_{Targ}$ nije dovoljna jer je zadata u apstraktnoj sintaksi $Targ$. Konkretizovanje sintakse se postiže pisanjem funkcije $targ2str$. Izračunavanjem izraza:

$$targ2str\ compiler_{Targ}$$

dobija se niska koja predstavlja izvršiv kôd napisan u jeziku $Targ$, za koji već važi da predstavlja verifikovan algoritam kompilacije. Korišćenjem komande dokazivača za prikazivanje vrednosti izraza[6] se dobija kôd koji je moguće iskoristiti van dokazivača. Ovime je na konceptualnom nivou zaokružen ceo postupak postizanja samokompilirajućeg prevodioca u okviru IDT.

[6] U sistemu *Coq* je to komanda `Print`.

Uz rad [14], na zvaničnoj stranici repozitorijuma dokazivača *HOL4* u okviru direktorijuma sa primerima se nalazi kompletna implementacija namenjena za korišćenje uz *GNU*-ov asembler. Iako je pristup pri dizajniranju bio takav da podržani jezici budu minimalni kako bi se prikazao koncept samokompilacije, broj linija u skriptama koje sadrže kôd je 7648.

Glava 5

Realizacija

U ovoj glavi će biti prikazano kako je moguće iskoristiti neke od tehnika koje su predstavljene u prethodnim poglavljima. Biće definisana dva jezika, od kojih ulazni sadrži sve praktične elemente pominjane u glavi o konstrukciji jezika. Izlazni jezik će podsećati na stek mašinu, ali bez namere da oslikava realistični jezik niskog nivoa. Za oba jezika će biti preskočeno razmatranje konkretne sintakse, samim tim i potreba za parserom. Generisanje koda sa ulaznog na izlazni jezik će se direktno vršiti funkcijom u jeziku *Gallina*. Na primeru manjih jednostavnih funkcija će biti prikazano kako se mogu, skoro automatski, generisati čiste funkcije, ili ručno implementirati one sa sporednim efektima. U ovom slučaju promena stanja se odnosi samo na promenu u modelu standardnog ulaza i izlaza. Takvi primeri funkcija će zatim biti i izvršeni korišćenjem denotacione funkcije značenja. Ekvivalentnost između dva načina interpretiranja izraza će implicitno biti pretpostavljena. Prilikom uvođenja lema i dokazivanja tvrđenja kojima se generišu izrazi u ulaznom jeziku, koristiće se relacija kojom se definiše operaciona semantika ulaznog jezika.

5.1 Ulazni jezik `exp`

Jezik sa kojeg se prevodi u velikoj meri liči na prethodno opisani jezik `Exp`. Induktivni tip izraza `exp` ipak implementira i neka bitna proširenja. Sem podrške za konstante, promenljive, dodelu vrednosti promenljivama, primitivne operacije i uslove grananja, postoji i konstruktor za pozivanje funkcija:

```
Call (f_id : fname) (params_exp : list exp)
```

Važno unapređenje predstavlja tip `dec` kojim se omogućava definisanje sopstvenih funkcija. U kodu 5.1 su navedeni pojedini elementi ulaznog jezika, a u nastavku će biti dati i opisi za neke od njih. Funkcija `if_test` prihvata uslov poređenja `Less` ili `Equal` i za konkretne, semantičke vrednosti vraća vrednost tipa `(@result bool)`. Zarad praktičnog slučaja poređenja enkodirane vrednosti niske sa enkodiranom vrednošću terminirajućeg karaktera, dozvoljavaju se poređenja vrednosti `VPair` sa `VNum`. Takvo ponašanje odgovara pravom jeziku Lisp, a kasnijim primerom, koji se tiče funkcije štampanja, biće razjašnjena potreba za tim.

Sa `lookup_fun` se postiže pretraga opisana prilikom navođenja teoreme (4.5). Pomoćna funkcija `make_env` se poziva jedino iz `env_and_body` i to za prazno okruženje – što i sam naziv akumulirajućeg parametra `acc` sugeriše. Uloga `make_env` je da dodeli vrednosti identifikatorima promenljivih – vraćajući odgovarajuće okruženje. Iz funkcije `env_and_body` se to okruženje vraća zajedno sa telom funkcije, prethodno dobijenim pozivanjem `lookup_fun`. Na ovaj način se simulira lokalno okruženje funkcija jezika `exp`.

```
Definition init_state (decls : list dec) (input : string) : state
Definition append_to_output (str : string) (s : state)
Definition remove_first (s : state) : state
Inductive error_type := Crash | TimeOut | Placeholder | Verbose (s : string).
Inductive result {A : Type} := Res (v : A) | Error (et : error_type).
Definition Success := VNum 0.
Definition eval_op (o : op) (args_val : list val) (s : state)
Definition if_test (t : test) (vs : list val)
Fixpoint lookup_fun (n : fname) (decls : list dec)
Fixpoint make_env (args_val : list val) (args_id : list vname) (acc : env)
Definition env_and_body (n : fname) (args_val : list val) (s : state)
Fixpoint eval' (t : nat) (e : env) (x : exp) (s : state)
Definition allowed_rec_depth : nat := 1000.
Definition eval (e : env) (x : exp) (s : state)
```

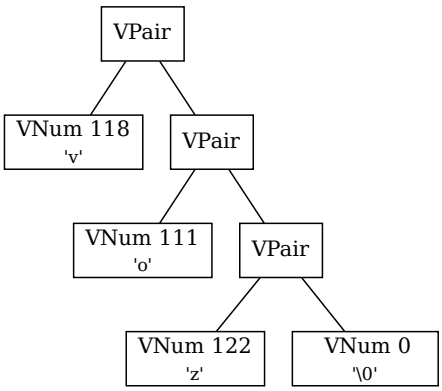
Kôd 5.1: Pregled bitnijih elemenata ulaznog jezika `exp`.

Funkcija `eval` daje denotaciono značenje: za datu uređenu trojku okruženja, izraza i polaznog stanja, vraća semantičku vrednost i završno stanje. U poglavlju 2.5, kada su opisivani neki od elemenata sistema *Coq*, spomenuto je da rekurzivna izračunavanja `Fixpoint` imaju strog sintaksički uslov smanjivanja po barem jednom

parametru. Zato se funkciji `eval_` dodaje parametar `(t : nat)`, kojim se traženi uslov ispunjava. Kako pri njenom pozivu vrednost tog argumenta biva proizvoljno odabrana, ona se sakriva od pozivaoca krajnjom funkcijom `eval`. Tako se pozivom `eval` zapravo samo poziva `eval_` sa dodatnim argumentom `allowed_rec_depth`. Implementacijom ove funkcije se omogućava pisanje konkretnih programa.

5.2 Primeri funkcija u jeziku `exp`

U nastavku će detaljno biti predstavljen način korišćenja ovog jezika, najpre pisanjem funkcije štampanja. Prikazom slučaja prevođenja rekurzivne funkcije *even*, u teoremi 4.7 su se pojavili simboli koji su označavali enkodirane vrednosti nekih niski. Tom prilikom je opisana funkcija koja se u ovoj implementaciji naziva `str_to_nat`. Takav način enkodiranja je pogodan za identifikatore, pošto je okruženje implementirano preko parcijalnih mapa koje slikaju iz `nat` u `nat`. Za skladištenje vrednosti niski u semantičkom domenu `Val` potrebno je izvršiti i njihovo enkodiranje. Enkodiranje pojedinačnih karaktera odgovara vrednosti iz *ASCII* tablice, koja je prosleđena konstruktoru `VNum`. Prazna niska se predstavlja kao stablo koje ima samo koren sa vrednošću koja odgovara terminirajućoj nuli. Slučaj neprazne niske se enkodira tako što, krećući se od korena, prvi karakter niske sleva biva enkodiran u levom detetu trenutnog čvora, a ostatak niske se počevši od desnog deteta trenutnog čvora enkodira na isti način. Rezultat je bijektivno preslikavanje niske u koso binarno stablo. Na slici 5.2 je dat primer rezultujućeg stabla. Tako implementirana funkcija je nazvana `str_to_val`.



Slika 5.2: Vrednost niske „voz” u tipu `Val`.

S ciljem da se demonstriraju koraci pri implementaciji funkcije štampanja, najpre je potrebno naglasiti da u do sad modeliranom jeziku postoje dva načina za čuvanje vrednosti neke niske. Jedan je da se napiše funkcija koja bi čitala neki broj karaktera sa ulaza primitivnom funkcijom `Op Read`, karakter po karakter i to odmah ispisivala ili na izlaz pomoću `Op Write`, ili u neku promenljivu korišćenjem `Op Cons`. Drugi način je da se funkciji prosleđuje `exp` vrednost, u vidu argumenta i da se ta vrednost štampa. Cela implementacija u jeziku Lisp je data kodom 5.3. Ispostavlja se da je implementaciju u `exp` moguće opisati na isti način.

```
(defun print (s)
  (if (= s '0) '0
    (let (tmp (write (head s)))
    (print (tail s)))))
```

Kôd 5.3: Implementacija funkcije štampanja iz [14], u nekom dijalektu Lisp-a.

```
Definition print_body_exp : exp :=
  If Equal [Var S; Const 0]
    (Const 0)
    (Let TMP (Op Write [Op Head [Var S]])
             (Call PRINT [Op Tail [Var S]])).

Definition print_dec : dec := Defun PRINT [S] print_body_exp.
```

Kôd 5.4: Funkcija `PRINT` jezika `exp`.

Bez eventualnog učitavanja posebnih modula koji podržavaju efikasniji rad sa aritmetičkim operacijama u *Coq*-u, modulima standardne biblioteke se veoma lako mogu preći granice maksimalnih vrednosti[1]. Iz tog razloga, umesto da identifikatori `vname` i `fname` zaista enkodiraju pune nazive identifikatora iz jezika *Gallina*, enkodiraće prvo, ili prva dva slova, što je u slučaju implementiranja malog jezika više nego dovoljno. Neka su date naredne definicije identifikatora:

```
Definition S : vname := str_to_nat "S".
Definition TMP : vname := str_to_nat "T".
```

[1]Izračunavanje 16-bitne vrednosti `nat` rezultuje ispisom greške `"Error: Stack overflow"`.

```
Definition PRINT : fname := str_to_nat "P".
```

Tada se funkcija štampanja može implementirati kodom 5.4, a kao u kodu 5.5, sa `Compute` se zadaje instrukcija *Coq*-u da izvrši izračunavanje izraza. U ovom slučaju treba pozvati funkciju `eval` sa odgovarajućim argumentima.

```
Compute eval
  (TMP |-> (str_to_val "Hello World!"))
  (Call PRINT [Var TMP])
  (init_state [print_dec] "").
```

Kôd 5.5: Interpretiranje konkretnog poziva funkcije PRINT.

Prethodnim kodom se za `exp` izraz čija je interpretacija jednaka interpretaciji niske `"Hello world!"` izvršava funkcija štampanja na standardni izlaz koji je modelovan poljem stanja programa. Izračunavanjem tog izraza se dobija izlaz dat u kodu 5.6.

```
= (Res (VNum 0),
  { decls := [Defun 80 [83]
               (If Equal [Var 83; Const 0] (Const 0)
                (Let 84 (Op Write [Op Head [Var 83]])
                 (Call 80 [Op Tail [Var 83]])))];
    input := "";
    output := "Hello World!"
   }) : result * state
```

Kôd 5.6: Rezultat izračunavanja izraza iz koda 5.5.

U skladu sa prethodnim opisom `eval` funkcije, ona vraća uređen par:

- Interpretacije rezultata – uspešno izvršavanje označeno nulom: `Res (VNum 0)`
- Završnog stanja predstavljenog kao slog sa tri polja:
  1. polje je lista deklaracija koja sadrži samo funkciju `PRINT`, a ovim ispisom se može napraviti uvid u internu reprezentaciju vrednosti identifikatora
  2. polje je ostalo netaknuto jer na ulazu nisu zadati podaci
  3. polje sadrži nisku `"Hello world!"` koja je ispisana na izlaz

Na sličan način, u svrhu demonstracije mogućnosti opisivanja izračunavanja u tipu `exp`, implementirane su još dve funkcije. Po uzoru na pokušaj implementacije funkcije `even` iz koda 2.7, implementacija u `exp` se dobija skoro pa direktnim prepisivanjem – što se vidi u bloku koda 5.7. Kao i u prethodnom slučaju, ovu funkciju je moguće pozvati na sledeći način:

```
Compute eval empty_env (Call EVEN [Const 4]) empty_with_even_dec.
```

Time se dobija rezultat `(Res (VNum 1))`, a stanje ostaje nepromenjeno[2]. Na osnovu enkodiranja vrednosti bulovskog tipa – u ovom slučaju je vraćena istinitosna vrednost `true`. Pošto je prosleđen argument `(Const 4)`, to odgovara tačnom rezultatu. Nakon implementirane funkcije `EVEN`, neko bi mogao da se zapita kako bi to moglo da se odštampa u jeziku `exp`. Jednostavno pozivanje funkcije `PRINT` za povratnu vrednost od `EVEN` se za neparne brojeve uspešno izvršava. Razlog je to što se enkodirana vrednost za `false` podudara sa enkodiranom vrednošću terminirajućeg karaktera. Propratnih efekata, doduše, ne bi bilo. Na izlaz bi bio pokušan ispis prazne niske, što ne bi imalo nikakvog uticaja na promenu stanja. U slučaju poziva funkcije `EVEN` za parne brojeve, dolazi do greške zbog nepodudaranja očekivanog tipa argumenta funkcije `Op Tail`.

```
Definition even_body_exp : exp :=
  If Equal [Var N; Const 0]
    (Const 1)
    (Op Sub [Const 1;
             Call EVEN [Op Sub [Var N; Const 1]]]).

Definition even_dec : dec := Defun EVEN [N] even_body_exp.
```

Kôd 5.7: Funkcija `even`, data u jeziku `exp`.

Da bi se omogućio ispis svih cifara, a samim tim i bulovskih vrednosti koje su 0 ili 1, može se definisati posebna funkcija. Na prirodan način se od vrednosti cifre, dobija *ASCII* vrednost (sabiranjem sa kodom za karakter cifre nula). *Coq* poseduje funkcije `nat_of_ascii` i `ascii_of_nat` za konverziju u oba smera. Da bi funkcija štampanja uspešno prepisala željenu cifru na izlaz, neophodno je prethodno dobijenu vrednost enkodirati na način kako se enkodiraju niske. U kodu 5.8 je prikazana

[2]Kreirano prazno stanje koje poseduje samo `even_dec` u listi deklaracija, je zbog preglednosti sačuvano u promenljivu `empty_with_even_dec`, što će i nadalje biti praksa.

takva funkcija enkodiranja. Sve što je potrebno uraditi je nadovezati vrednost terminirajućeg karaktera, čime se dobija opis strukture niske sa jednim karakterom (odgovarajućom cifrom).

```
Definition ascii_0_to_exp := Const (nat_of_ascii "0").
Definition dts_body_exp : exp :=
  Op Cons [Op Add [Var M; ascii_0_to_exp]; Const 0].

Definition dts_dec : dec := Defun DTS [M] dts_body_exp.
```

Kôd 5.8: Funkcija kojom se mogu odštampati cifre na izlaz.

Konačno, iskorišćavanjem svih ovih funkcija moguće je za neko brojčano izračunavanje izraza jezika `exp` proveriti da li je parno ili neparno i tu vrednost odštampati na izlaz. Takav primer je ilustrovan kodom iz dodatka 8. Ideja je da se kompozicijom dovoljnog broja ovakvih funkcija postepeno izgrađuju komplikovanije funkcije. Na kraju bi se mogla napraviti i funkcija za leksiranje i parsiranje konkretne sintakse, što ovim radom neće biti pokriveno. Prethodnim primerima je pokazano da je moguće ručno napisati određene funkcije, ali ništa nije tvrđeno o njihovoj korektnosti. Svrha modelovanja programskih jezika u dokazivačima teorema se ogleda upravo u raspoloživosti mehanizama za sprovođenje formalnih dokaza korektnosti.

U prethodnim primerima bi za svaku od funkcija bilo potrebno dokazati teoremu kojom se univerzalno kvantifikuju i dovode u vezu određeni elementi od značaja. Bez dokaza, u kodu 5.9 dat je primer takve teoreme. Umesto `eval`, koja ima fiksiran opadajući argument, tvrđenje je formulisano korišćenjem `eval'`, uz neko dovoljno veliko `t`. U suprotnom bi konstanta `allowed_rec_depth` bila gornja granica broja koraka pri tumačenju izraza. Iz istog razloga, pri izračunavanju proizvoljne vrednosti izraza `x` (koji mora da bude broj), potrebno je odgovarajuće `tx`. Povratna vrednost oba izračunavanja treba da bude jednaka za svaki broj sa ulaza.

```
Theorem eval_EVEN_iff_even : forall e x n s, exists tx t,
  In even_dec s.(decls) ->
  eval' tx e x s = (Res (VNum n), s) ->
  eval' t e (Call EVEN [x]) s = (Res (Bool (even n)), s).
```

Kôd 5.9: Teorema kojom se tvrdi da `EVEN` ima ponašanje funkcije `even`.

Pošto je proces ručnog pisanja funkcija u modelovanom jeziku mukotrpan, kao i samo dokazivanje tvrđenja o njihovoj korektnosti, donekle automatizovani pristup iz naredne glave olakšava ta dva postupka, a njime se u nekoj meri praktično približava tehnika opisana u poglavlju 4.3, o rekurzivnom slučaju prevođenja.

5.3 Pomoć strategije *Ltac*-a

Da bi se napisala funkcija `eval` čiji je tip naveden u kodu 5.1, potrebno je posebno obraditi svaki mogući slučaj izračunavanja. Primera radi, u `eval_op` su implementirane sve operacije `Op`. Kada god je prosleđeni tip testa `Div` – samo ukoliko su argumenti oblika liste dve brojčane vrednosti `[VNum n1; VNum n2]`, tada ima smisla razmatrati operaciju. U ostalim slučajevima se obustavlja rad i prijavljuje se greška o nepoklapanju tipova. Međutim, samo izračunavanje u prvoj vrsti slučaja nije neposredni poziv operatora deljenja, već je potrebno dodatno proveriti da li se delilac razlikuje od nule.

Takođe, pojedine konstrukcije *Coq*-a su pogodnije za korišćenje u kasnijem dokazivanju. Ukoliko se prethodni uslov proverava korišćenjem naredbe `if`, to može u nekoj meri ili onemogućiti, ili produžiti dokaz, u odnosu na slučaj kada je doneta odluka da se za proveru koristi naredba `match`. Jedan takav slučaj, zajedno sa prikazom stanja interaktivnog dokazivača, biće dat u nastavku. Sve pobrojano ukazuje da je za dokazivanje tvrđenja koja sadrže složena izračunavanja potrebno određeno iskustvo u radu sa dokazivačem, kako bi se donosile odluke koje neće otežati dalje korake pri dokazivanju. Kako bi se olakšalo dokazivanje lema, u nastavku rada će se polaziti od semantike uvedene relacijom `eval_R`, bez prethodnog dokazivanja ekvivalentnosti sa `eval` semantikom.

```
Inductive eval_R : (env * (list exp) * state) -> ((list val) * state) -> Prop
Inductive app : fname -> list val -> state -> (val * state) -> Prop
```

Kôd 5.10: Prirodna operaciona semantika izraza tipa `exp`.

U kodu 5.10 data je operaciona semantika u vidu relacije `eval_R`, koja je nalik na induktivan tip `evalExp_R` iz ranijeg poglavlja. Relacija `eval_R` je u nastavku prikazivana notacijom `==>` zadatom u *Coq*-u. Njome se lista izraza tipa `exp`, zajedno sa nekim okruženjem i početnim stanjem, dovodi u vezu sa listom vrednosti tipa

`val` i završnim stanjem. Relacija `app` odgovara istoimenoj relaciji uvedenoj teoremom (4.3) i ona daje vezu između identifikatora funkcije, liste izraza tipa `val` (koji predstavljaju vrednosti prosleđenih argumenata) i uređenog para povratne vrednosti tipa `val` i završnog stanja.

Pokušaj jednostavne rekurzivne definicije funkcije `even` iz koda 2.7 na strani 15 nije bio uspešan zbog pomenutog ograničenja konstrukcije `Fixpoint`. Jedno moguće rešenje je korišćenje biblioteke *Equations* [20]. Ta biblioteka znatno olakšava rad sa rekurzivnim funkcijama. Međutim, s njom se uvodi specijalna sintaksa što otežava cilj prevođenja *Coq* izraza u tip `exp`. U slučaju jednostavne funkcije poput `even`, ugrađena komanda `Program` bi mogla biti dovoljna. Pokušaj njene upotrebe prikazan je u kodu 5.11, zajedno sa ispisom trenutnog stanja dokazivača, tj. konteksta i cilja.

```
Program Fixpoint even (n : nat) {measure (n)} : bool :=
  if 0 =? n then negb (even (n - 1)) else true.
Next Obligation. (* ... očekuje se dokaz zaustavljanja rekurzije *)
```

```
1 goal (ID 604)
  n : nat
  even : forall n0 : nat, n0 < n -> bool
  ============================
  n - 1 < n
```

Kôd 5.11: Pokušaj implementacije funkcije *even* korišćenjem `Program Fixpoint` konstrukta (gore) i ispis trenutnog stanja interaktivnog dokazivača (dole).

Novina u odnosu na prethodne definicije je u delu `{measure (n)}` iz potpisa. U njemu se navodi koja je to mera koja se zasigurno smanjuje u narednom rekurzivnom pozivu (a moguće je navesti i relaciju, za koju bi trebalo dokazati da je *dobro zasnovana [19]*). Dokaz koji mora da se pruži navodi se nakon ključnih reči `Next Obligation` i u ovom slučaju potrebno je dokazati da je `n-1 < n`, za svako `n`. To naravno nije tačno jer se posmatraju samo elementi iz `nat`, a korišćenjem `if` naredbe se ne čuva informacija o tome da je `n` zapravo različito od nule u rekurzivnom slučaju (pa je samim tim neophodni dokaz nejednakosti nemoguć). Moguće je uveriti se u to, proverom konteksta i cilja sa početka dokaza.

S obzirom da je reč o prirodnim brojevima, može se napisati ekvivalentna formulacija funkcije. Naime, nakon zamene operatora `=?` operatorom `<?`, bilo bi potrebno

samo obrnuti redosled izraza pri grananju. Bez zalaženja u detalje, operator `lt_dec`, vraća tip koji može da se koristi u uslovu komande `if`, umesto bulovskog operatora `<?`, tako da pri tom, u kontekst bude ubačena odgovarajuća hipoteza: `H : 0 < n`. Definicija funkcije *even* korišćenjem pomenutog operatora je navedena u kodu 5.12.

```
Program Fixpoint even (n : nat) {measure (n)} : bool :=
  if lt_dec 0 n then negb (even (n - 1)) else true.
Next Obligation.
destruct n ; lia. Defined.
```

Kôd 5.12: Implementacija funkcije *even* korišćenjem `Program Fixpoint`.

Nakon što je definisana funkcija koju treba prevesti u jezik koji se modeluje, može se pristupiti narednim koracima opisanim u poglavlju 4.3. U ovom master radu se neće vršiti obilazak *AST*-a izraza u *Coq*-u, zbog usložnjavanja postupka, a za kompletan postupak opisan u radovima [16] i [14] to jeste neophodno jer je tako moguće na potpuno opšti način parsirati svaki od podizraza odozdo-naviše (uključujući nazive identifikatora). Umesto toga, u nastavku je dat praktični pogled kroz koji se može steći slika o tome koji su sve koraci neophodni, sem obilaska *AST*-a. Funkcija enkodiranja bulovskih vrednosti `Bool`, leme `bool_true_lemma`, `bool_false_lemma` i neznatno promenjena[3] `if_less_lemma_b` su ranije opisivane, a biće neophodne za postupak koji sledi.

```
Lemma call_arg_decrement : forall (e : env) (n : nat) (s : state)
                  (v_id : vname) (f_id : fname) (f : nat -> bool),
 ((e, [Var v_id], s) ==> ([VNum n], s)) ->
 app f_id [VNum (n - 1)] s (Bool (f (n - 1)), s) ->
 (e, [Call f_id [Op Sub [Var v_id; Const 1]]], s) ==> ([Bool (f (n - 1))], s).
Proof.
 intros; eapply app_call. (* lema koja povezuje relacije app i eval_R *)
 repeat (econstructor; eauto). assumption.
Qed.
```

Kôd 5.13: Lema za dekrementiranje argumenata unarne funkcije.

[3]Uz suštinski iste pretpostavke, samo prilagođene svim elementima ovog jezika, iskaz koji se tvrdi sadrži operator `lt_dec`, jer je to ono što će da se prevodi, a ne operator `<?`.

Posmatranjem tela funkcije `even`, može se uočiti da prilikom rekurzivnog poziva argument biva dekrementiran. Jednostavne operacije poput te mogu, na manje ili više opšt način, biti uvedene nekom lemom. Na primer, u slučaju dekrementiranja, dovoljno je posmatrati unarne funkcije, umesto da se, recimo, posmatra binarna operacija oduzimanja. Za potrebe demonstracije, u kodu 5.13 je navedena lema sa dokazom.

S obzirom na to da obrađivanje naziva promenljivih nije automatski prosleđeno funkciji enkodiranja, sledeći korak podrazumeva uvođenje novog identifikatora – za funkciju čije će telo biti generisano:

```
Definition EVEN' : fname := str_to_nat "E'".
```

Kao što je spomenuto u poglavlju 4.3, da bi novouvedena oznaka dobila neko značenje koje ima veze sa funkcijom `even`, privremeno se dodaje aksioma:

```
Axiom evenA : forall s n, app EVEN' [VNum n] s (Bool (even (n)), s).
```

Dodavanjem strategije *Ltac* (imenovane `use_lemmas_auto`), koja sadrži pomenute leme i aksiomu, dokazuje se tvrđenje dato u kodu 5.14.

```
Lemma gen_EVEN : forall (e : env) (n : nat) (s : state),
                                 exists (even_body : exp),
  e N = Some (VNum n) ->
  app EVEN' [VNum (n - 1)] s (Bool (even (n - 1)), s) ->
  (e, [even_body], s) ==>
    ([Bool (if lt_dec 0 n then negb (even (n - 1)) else true)], s).
Proof. repeat use_lemmas_auto. Qed.
Print gen_EVEN.
```

Kôd 5.14: Teorema kojom se dobija kôd tela funkcije `EVEN` za konkretan poziv.

Ovakav način u maloj meri ubrzava ručni proces dokazivanja, ali sa porastom broja funkcija koje se na ovaj način generišu, mogu se čuvati i leme koje su korišćene za njihovo generisanje, pa bi na taj način, vremenom, proces generisanja mogao biti sve jednostavniji. Problem svakako jeste u smeru kretanja odozgo-naniže. Kada bi obilazak bio iz suprotnog smera i nazivi identifikatora bili dostupni, proces generisanja novih funkcija bi zasigurno bivao sve brži. Generisana vrednost `even_body`, ispisana je `Print` komandom i ovde se navodi u bloku koda 5.15.

```
(If Less [Const 0; Const n]
  (Op Sub [Const 1; Call EVEN' [Op Sub [Var N; Const 1]]])
  (Const 1))
```

Kôd 5.15: Deo rezultata koji se dobija pozivom funkcije `Print` iz koda 5.14.

Prilikom pokušaja interpretacije dobijenog koda, funkcijom `eval`, dokazivač bi prijavio grešku, ispisavši da promenljiva `n` nije definisana. Iz razloga nedostupnosti naziva promenljive, ona nije enkodirana funkcijom `str_to_val` i potrebno je ručno uočiti mesta gde se u `even` funkciji pojavljuje promenljiva – kao i u slučaju dodavanja leme za dekrementiranje. Time dokazivač olakšava pronalaženje neispravno prevedenih elemenata. U ovom slučaju dovoljno je zameniti `Const n` iz uslova grananja sa `Var N` i dobijeni kôd zaista odgovara funkciji `eval`.

5.4 Ciljni jezik `targ`

U ovom poglavlju će ukratko biti prikazan i jezik nižeg nivoa koji podseća na stek mašinu. Realizacija je delimično odrađena po uzoru na jednostavan primer iz drugog poglavlja knjige [5]. Kako bi se omogućilo potpuno preslikavanje iz `exp` u takav jezik `targ`[4], biće načinjene neke pretpostavke o obliku `exp` izraza koji se prevode. Tipovi izraza s kojima će biti pretpostavljen dalji rad su oni `exp` izrazi koji se interpretiraju kao brojčane vrednosti ili liste (pa i niske). Identifikatori iz prethodnog poglavlja će umesto prirodnim brojem, biti predstavljeni induktivnim tipom sa konstruktorima bez parametara, koji imaju ulogu nabrojivog tipa. Konstruktori za `vname` su nazvani `VM`, `VN`, `VO` i `VS` i tim redom odgovaraju registrima `RX`, `RY`, `RZ`, `RL`. Tom promenom se suštinski ne menjaju funkcionalnosti prethodno definisanog polaznog jezika. Funkcije su predefinisane u oba jezika. Poziv jedne funkcije u `exp` se direktno preslikava u poziv odgovarajuće funkcije u `targ`. Ovim pretpostavkama se olakšava preslikavanje identifikatora i time se pojednostavljuje funkcija prevođenja iz `exp` u `targ`.

Memorija može biti modelovana slogom koji čuva registre, a zarad jednostavnosti, neka su tim slogom modelovani i ulaz i izlaz, u vidu niske. Najpre, stek se može posmatrati kao lista prirodnih brojeva. Uobičajeno nazvanim instrukcijama se dodaju vrednosti iz registra na stek i obrnuto. Operandi se uzimaju sa vrha steka,

[4]U glavi 4 o samokompilaciji, to preslikavanje se naziva *codegen*.

izuzev kod instrukcija čitanja sa ulaza i guranja vrednosti na stek. Takođe, sem kod instrukcije za pisanje u registar, po obavljenoj operaciji rezultat se dodaje na stek. Program je definisan kao lista instrukcija. Semantika instrukcije se zadaje funkcijom:

```
Definition instrDenote (i : instr) (s : stack) (mem : memory) :
                (option stack) * (option memory)
```

Za datu instrukciju `i`, neko trenutno stanje steka i trenutni presek vrednosti u memoriji (uključujući ulaz i izlaz), ukoliko nema greške pri izvršavanju instrukcije, vraća se par koji odgovara novom stanju steka i memorije, a u suprotnom se kao jedna od koordinata vraća vrednost `None`.

Treba napomenuti da one instrukcije koje utiču na promenu toka izvršavanja nema smisla tumačiti izolovano od ostatka programa. Takve instrukcije su `iLess`, `iEqual` i `iFunCall`. Prve dve odgovaraju uslovu grananja, koji se ispituje nad dva broja sa vrha steka. Prilikom tumačenja ovih dveju instrukcija, odbacuje se jedan od dva potprograma, odnosno jedna od dve podliste koje odgovaraju *then* i *else* grani.

Interpretacija kompletnog programa se postiže funkcijom:

```
Fixpoint targDenote (p : targ) (s : stack) (mem : memory) :
                (option stack) * (option memory)
```

Obilaskom cele liste se na odgovarajući način interpretira jedna po jedna instrukcija, pomoću funkcije `instrDenote`, sem u slučaju `iLess`, `iEqual` i `iFunCall`, koje se obrađuju posebno. Prilikom interpretacije naredne instrukcije se uzima ažurirana vrednost steka i memorije, nakon što tekuća bude izvršena. Po završetku programa se vraća krajnja vrednost steka i memorije.

Primer izračunavanja u `targ`

Neka je instrukcijama jezika `targ` potrebno opisati naredno izračunavanje. Maksimum vrednosti registara `RY` i `RZ` privremeno smestiti u registar `RX` i izračunati zbir sva tri registra, a zatim za prvobitnu vrednost registra `RX` proveriti da li je parna i rezultat (`"1"` ili `"0"`) ispisati na izlaz.

Instrukcijama koje su na raspolaganju, odgovarajući program počinje guranjem vrednosti registra `RX` na stek. To se postiže instrukcijom `iPushReg X` i time se vrednost čuva za kasnije korišćenje. Zatim se, za svrhu pronalaženja maksimuma, na stek guraju vrednosti `RY` i `RZ`, i instrukcijom za uslov grananja `iLess` se bira veća od te dve vrednosti. Sabiranje tri promenljive se vrši ponovljenim korišćenjem operacije

sabiranja `iArith iPlus`, nakon guranja vrednosti registara na stek. Time je opisan prvi deo izračunavanja – sa ažuriranom vrednošću registra `RX`.

Vrednost sa vrha steka se postavlja u registar `RX` instrukcijom `iMov X`. Slično kao i u slučaju `Op Write` operacije iz jezika `exp`, instrukcija za pisanje `iWrite` dohvata broj sa vrha steka i tumačeći ga kao vrednost iz *ASCII* tabele, ispisuje na izlaz. Zato je u slučaju modelovanja bulovske funkcije, na vrednost dobijenu nakon poziva (1 ili 0), potrebno dodati brojčanu vrednost *ASCII* koda cifre 0. Poziv funkcije koja ispituje parnost broja postiže se instrukcijom `iFunCall fEVEN`. Za svaku funkciju jezika `exp`, koja se prevodi u funkciju jezika `targ`, zna se koje se promenljive koriste lokalno, pa se u tom slučaju na sličan način obezbeđuje lokalno korišćenje registara koji odgovaraju tim promenljivama. U slučaju funkcije `fEVEN`, registar `RY` odgovara promenljivoj koja se koristila pri pravljenju `dec` vrednosti za funkciju sa identifikatorom `EVEN` (u pitanju je `VN`). Funkcijom koja prevodi, tokom izvršavanja se generišu instrukcije koje odgovaraju već unapred poznatom telu za `EVEN`, čime se omogućava vršenje rekurzivnih poziva. Po završetku rada ove funkcije, čiji je tip povratne vrednosti broj, rezultat se upisuje u registar `EAX`, a zatim se izlazi iz lokalnog okruženja promenljivih tako što se vrednost gurnuta na stek pre poziva funkcije sada premešta nazad u registar `RY`. Ovime je ugrubo objašnjeno kako je moguće na jeziku `targ` napisati zahtev iz početnog pasusa – konkretan program je u kodu 5.16.

```
[ iPushReg X;     (* sačuvana vrednost RX *)
  iPushReg Y; iPushReg Z; iLess; iPushReg Z; iNoopThen;
  iPushReg Y; iNoopElse; iMov X; (* RX := max(RY, RZ) *)
  iPushReg X; iPushReg Y; iPushReg Z; iArith iPlus; iArith iPlus;
  iMov AX;        (* EAX := RX + RY + RZ *)
  iMov X;         (* vraćena vrednost RX *)
  iPushReg AX;    (* vrednost izračunavanja gurnuta na stek *)
  iPushReg Y;     (* sačuvana vrednost RY pre poziva funkcije *)
  iPushReg X; iMov Y; iFunCall fEVEN;
  (* ... ovde se tokom izvršavanja umeće prevod tela funkcije EVEN *)

  iMov AX;                (* EAX := fEVEN(RX) *)
  iMov Y; iPushReg AX;
  iPushConst 48;          (* vrednost (nat_of_ascii "0") gurnuta na stek *)
  iArith iPlus; iWrite ] (* ispis "1" ili "0" *)
```

Kôd 5.16: Primer programa u jeziku `targ` kojim se prikazuje upotreba registara za neka izračunavanja. Parnost početne vrednosti registra `RX` se ispisuje na izlaz.

Prethodni blok koda je dobijen korišćenjem funkcije `codegen`, prevođenjem odgovarajućih izraza tipa `exp`. Stanje steka i memorije nakon pokretanja prethodnog programa funkcijom `targDenote`, sa inicijalno postavljenim vrednostima za `RX`, `RY` i `RZ` na 2, 3 i 4, dato je u bloku koda 5.17. Može se primetiti da je na vrhu steka 0, zbog uspešnog izvršavanja operacije pisanja karaktera, a 11 je prethodno dobijeni zbir vrednosti registara.

```
= (Some [0; 11], Some {| RX := 2; RY := 3; RZ := 4; RL := [];
                         EAX := 1; EAXL := []; input := ""; output := "1" |})
: option stack * option memory
```

Kôd 5.17: Rezultat nakon izvršavanja programa iz koda 5.16.

Funkcija prevođenja `codegen` prosleđenu listu izraza tipa `exp` prevodi u `targ`. Do sada nije precizno prikazano kako se vrši umetanje koda koji opisuje telo funkcije koja se prevodi. Na primeru prevoda funkcije `[Call EVEN [Const 1]]`, dobijenog pozivom `codegen`, u dodatku 9 prikazana su bitnija stanja kroz koja tok izvršavanja funkcije `targDenote` prolazi pri izračunavanju instrukcije `iFunCall`. Jednostavniji primer (bez poziva funkcija) upotrebe `codegen` i `targDenote`, dat je u kodu 5.18.

```
Definition swap_e : exp :=
 (Let VM (Op Add [Var VM; Var VN])
    (Let VN (Op Sub [Var VM; Var VN])
       (Let VM (Op Sub [Var VM; Var VN])
          (Op Add [Op Mult [Const 10; Var VM]; Var VN])))).
Definition swap_t : targ := codegen [swap_e].
Compute swap_t. (* =
[iPushReg X; iPushReg X; iPushReg Y; iArith iPlus; iMov X; iPushReg Y;
  iPushReg X; iPushReg Y; iArith iSub; iMov Y; iPushReg X; iPushReg X;
  iPushReg Y; iArith iSub; iMov X; iPushConst 10; iPushReg X; iArith iMult;
  iPushReg Y; iArith iPlus; iMov AX; iMov X; iPushReg AX; iMov AX; iMov Y;
  iPushReg AX; iMov AX; iMov X; iPushReg AX] : targ  *)
Definition init_m : memory := ((input_memory "") < RX := 2 > < RY := 3 >).
Compute targDenote swap_t [] init_m. (* =
(Some [32], Some { RX := 2; RY := 3; RZ := 0; RL := []; EAX := 32; ... })
: option stack * option memory *)
```

Kôd 5.18: Primer upotrebe funkcija `codegen` i `targDenote`.

Teoremom o ispravnosti generatora koda 4.6, na strani 34, tvrdi se da je funkcija prevođenja iz jednog u drugi jezik korektna ukoliko (za isti ulaz) proizvoljan izraz u prvom jeziku proizvodi isti izlaz kao i njegov prevod na drugi jezik. U kodu 5.19, za jezike `exp` i `targ`, prikazano je kako može da izgleda takva teorema, bez dokaza. Njome se tvrdi da funkcija `codegen` dobro preslikava iz tipa `exp` u tip `targ`. Jači oblik tvrđenja bi mogao da uključi i jednakost između povratnih vrednosti.

```
Theorem codegen_correct : forall e s m x v state' stack' mem', exists t,
   sameInput_R e s m ->
   eval' t e x s = (Res v, state') ->
   targDenote (codegen [x]) [] m = (Some stack', Some mem') ->
   Exp.output state' = Stack.output mem'. Admitted.
```

Kôd 5.19: Teorema o ispravnosti `codegen` funkcije.

5.5 Preostali koraci

Ovim delom rada je dat pregled dodatnih koraka koje je potrebno načiniti kako bi se došlo do prve izvršive verzije samokompilatora. Najzahtevniji zadatak koji je potrebno obaviti je potpuna automatizacija prevođenja čistih funkcija iz jezika *Gallina* u jezik `exp`, pristupom odozdo-naviše. Time bi se eliminisao dodatan korak dokazivanja ispravnosti manjih čistih potfunkcija funkcije `compiler` (napisane u jeziku *Gallina*) koja parsira konkretnu sintaksu. U ovom slučaju konkretna sintaksa bi se minimalno razlikovala od apstraktne – svi izrazi moraju biti oivičeni zagradama, a simbol za operaciju se navodi iza otvorene zagrade. Nakon dobijanja funkcije za parsiranje u jeziku `exp`, u oznaci `compiler_exp`, a zbog potfunkcija koje nisu čiste, potrebno je dokazati da `compiler_exp` i `compiler` daju isti izlaz, za proizvoljan ulaz. Dokazivanjem teoreme o ispravnosti generatora koda, poput one iz koda 5.19, dobija se oblik funkcije parsiranja napisan u jeziku `targ`, što bi pod pretpostavkom da je `targ` zaista izvršiv, značilo da je moguće sprovesti samokompilaciju.

Umesto smišljanja konkretne sintakse za `targ`, koji je dat kao lista unapred poznatih instrukcija, program je moguće posmatrati kao listu tokena i zatim implementirati funkciju koja se ponaša kao `targDenote`. Jedna mogućnost da se ovaj postupak ubrza je da se iskoristi `targDenote`. Implementiranjem funkcije u jeziku *Gallina* (tipa `string -> instr`) koja instrukciju zadatu kao niska (tipa `string`)

preslikava u *zaista* tu instrukciju (tipa `instr`). Tako bi mogao da se iskoristi *Coq*-ov mehanizam ekstrakcije koda i dobije izvršiva datoteka koja interpretira programe u jeziku `targ`.

Glava 6

Zaključak

Prilikom razvijanja netrivijalnog softvera, proveravanje i kompilatora i sopstvenog koda svaki put kada se uoči neka greška tokom razvoja, nametalo bi dodatnu obavezu programeru i usporavalo projekat. Zbog određenog nivoa podrazumevanog poverenja u kompilator, greške prouzrokovane neispravnostima u prevodiocu mogu biti teže uočljive u odnosu na greške u izvornom kodu za koji se taj prevodilac koristi.

Ovim radom je data osnova za verifikaciju prevodilaca korišćenjem IDT-a i konceptualno je opisan način na koji je postignuto verifikovanje zapaženog primera samokompilatora *CakeML*. Uporedo sa opštim okvirom takvog vida verifikacije, davani su isečci koda kojima se približavaju, uglavnom načini modelovanja određenih jezičkih konstrukata. Uz to su date smernice i kroz više primera tvrđenja, mogao se steći uvid u postupak davanja instrukcija *Coq*-u, kojima se dobija njihov dokaz. Takođe, pokazan je moguć način implementiranja jezika koji podržava kreiranje i poziv funkcija sa lokalnim okruženjem promenljivih. Treba napomenuti da implementirani jezici ni na koji način ne pretenduju da poseduju realistične detalje kojima se, na primer, razmatraju aspekti memorijskog modela, garancije zaustavljanja nakon pokretanja, ili paralelnog izvršavanja. U delu koji se tiče samokompilacije je analiziran pristup postupka automatizovanog dokazivanja kojim se uz dobijanje korektnosti koda generisanog u ulaznom jeziku, dobija i dokaz da je time semantika zadržana u odnosu na odgovarajuću funkciju metajezika dokazivača. Praktični aspekt takve tehnike je razmatran, ali ne i implementiran, pa bi se unapređenje ovog rada pre svega ogledalo u tome. Pribegavanje naprednijim tehnikama funkcionalnog stila programiranja i korišćenje dodataka poput *Coq-Elpi* [21], bi za tu svrhu moglo obezbediti, u ovom radu, izostavljene elemente.

Bibliografija

[1] AbsInt. CompCert: formally verified optimizing C compiler. `https://www.absint.com/compcert/index.htm`.

[2] Adam Chlipala. *Formal Reasoning About Programs*. In-progress, open-source book.

[3] Adam Chlipala. Skipping the binder bureaucracy with mixed embeddings in a semantics course (functional pearl). In *Proceedings of the 26th ACM SIGPLAN International Conference on Functional Programming (ICFP'21). August 2021.*

[4] Adam Chlipala. A verified compiler for an impure functional language. In *POPL '10*, 2010.

[5] Adam Chlipala. *Certified programming with dependent types*. MIT Press, April 21, 2019.

[6] The Coq development team. *The Coq proof assistant reference manual*. LogiCal Project. Version 8.15.2.

[7] Filip Marić, Matematički fakultet, Univerziteta u Beogradu. A survey of interactive theorem proving, 2015.

[8] Scott Owens Guodong Li and Konrad Slind. Structure of a proof-producing compiler for a subset of higher order logic. In *Programming Languages and Systems*, pages 205–219, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[9] UK John Wickerson, Paul Brunet, Imperial College London. Pearl: Diagrams for composing compilers, 2012.

[10] Xavier Leroy. The compcert c verified compiler, documentation and user's manual, version 3.11.

[11] Xavier Leroy. The formal verification of compilers (course), deepspec summer school 2017.

[12] Milena Vujošević Janičić, Matematički fakultet, Univerziteta u Beogradu. Materijali i slajdovi sa predavanja. `http://www.verifikacijasoftvera.matf.bg.ac.rs/vs/predavanja/06_semantika/06_semantika.pdf`.

[13] Robin Milner and R.W. Weyhrauch. Proving compiler correctness in a mechanised logic. *Machine Intelligence*, 7:51–73, 1972.

[14] Magnus O. Myreen. A minimalistic verified bootstrapped compiler (proof pearl). In *CPP '21: 10th ACM SIGPLAN International Conference on Certified Programs and Proofs, Virtual Event, Denmark, January 17-19, 2021*.

[15] Magnus O. Myreen. Verified compilers, compiler construction (course), spring 2019.

[16] Magnus O. Myreen. Functional programs: Conversions between deep and shallow embeddings, 2012.

[17] Magnus O. Myreen. The CakeML project's quest for ever stronger correctness theorems. In *Interactive Theorem Proving (ITP)*. LIPIcs, 2021.

[18] Benjamin C. Pierce, Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, and Brent Yorgey. *Logical Foundations*, volume 1 of *Software Foundations*. Electronic textbook, 2021. `http://softwarefoundations.cis.upenn.edu`.

[19] Predrag Janičić, Filip Marić. *Programiranje 2, Osnove programiranja kroz programski jezik C*. Matematički fakultet Univerziteta u Beogradu.

[20] Matthieu Sozeau and Cyprien Mangin. Equations - a function definition plugin. `https://github.com/mattam82/Coq-Equations`.

[21] Enrico Tassi. Elpi: an extension language for Coq (Metaprogramming Coq in the Elpi λProlog dialect), 2018.

[22] Ken Thompson. Reflections on trusting trust. *Commun. ACM*, 27(8):761–763, aug 1984.

[23] Yang, Xuejun and Chen, Yang and Eide, Eric and Regehr, John. Finding and Understanding Bugs in C Compilers. PLDI '11. ACM, 2011.

Prilozi

```
Definition eval_op (op : PrimOp) (params : list (option Val)) : option Val :=
  match op, params with
  | _, [None] | _, [None; None] => None
  | Add, [Some (VNum n1); Some (VNum n2)] => Some (VNum (n1 + n2))
  | Sub, [Some (VNum n1); Some (VNum n2)] => Some (VNum (n1 - n2))
  | Mult, [Some (VNum n1); Some (VNum n2)] => Some (VNum (n1 * n2))
  | Div, [Some (VNum n1); Some (VNum 0)] => None
  | Div, [Some (VNum n1); Some (VNum n2)] => Some (VNum (div n1 n2))
  | Cons, [Some v1; Some v2] => Some (VPair v1 v2)
  | Head, [Some (VPair h t)] => Some (h) (* odsecanje desnog stabla *)
  | Tail, [Some (VPair h t)] => Some (t) (* odsecanje levog stabla *)
  | _, _ => None
```

Prilog 1: Primer koda funkcije `eval_op` koja se koristi u kodu 3.3.

```
  Inductive evalExp_R : Exp -> Val -> Prop :=
  | E_Const (n : nat) : (Const n) ==> (VNum n)
  (* Sekcija za "Op" slučajeve. *)
  | E_OpAdd (params : list Exp) (e1 e2 : Exp) (n1 n2 : nat) :
    e1 ==> (VNum n1) -> e2 ==> (VNum n2) ->
    params = [e1; e2] ->
    Op Add params ==> VNum (n1 + n2)
  | E_OpSub (params : list Exp) (e1 e2 : Exp) (n1 n2 : nat) :
    e1 ==> (VNum n1) -> e2 ==> (VNum n2) ->
    params = [e1; e2] ->
    Op Sub params ==> VNum (n1 - n2)
  | E_OpMult (params : list Exp) (e1 e2 : Exp) (n1 n2 : nat) :
    e1 ==> (VNum n1) -> e2 ==> (VNum n2) ->
    params = [e1; e2] ->
```

```
    Op Sub params ==> VNum (n1 * n2)
  | E_OpDiv (params : list Exp) (e1 e2 : Exp) (n1 n2 : nat) :
    n2 <> 0 ->
    e1 ==> (VNum n1) -> e2 ==> (VNum n2) ->
    params = [e1; e2] ->
    Op Div params ==> VNum (n1 / n2)
  | E_OpCons (params : list Exp) (e1 e2 : Exp) (v1 v2 : Val) :
    e1 ==> v1 -> e2 ==> v2 ->
    params = [e1; e2] ->
    Op Cons params ==> VPair v1 v2
  | E_OpHead (params : list Exp) (e : Exp) (v1 v2 : Val) :
    e ==> (VPair v1 v2) ->
    params = [e] ->
    Op Head params ==> v1
  | E_OpTail (params : list Exp) (e : Exp) (v1 v2 : Val) :
    e ==> (VPair v1 v2) ->
    params = [e] ->
    Op Tail params ==> v2
  (* Sekcija za "If" slučajeve. *)
  | E_IfLessThen (params : list Exp) (e1 e2 : Exp) (n1 n2 : nat)
      (then_exp else_exp : Exp) (v : Val):
    n1 < n2 ->
    e1 ==> (VNum n1) -> e2 ==> (VNum n2) ->
    params = [e1; e2] ->
    then_exp ==> v ->
    (If Less params then_exp else_exp) ==> v
  | E_IfLessElse (params : list Exp) (e1 e2 : Exp) (n1 n2 : nat)
      (then_exp else_exp : Exp) (v : Val):
    n2 <= n1 ->
    e1 ==> (VNum n1) -> e2 ==> (VNum n2) ->
    params = [e1; e2] ->
    else_exp ==> v ->
    (If Less params then_exp else_exp) ==> v
  | E_IfEqThen (params : list Exp) (e1 e2 : Exp) (n : nat)
      (then_exp else_exp : Exp) (v : Val):
    e1 ==> (VNum n) -> e2 ==> (VNum n) ->
    params = [e1; e2] ->
    then_exp ==> v ->
    (If Equal params then_exp else_exp) ==> v
  | E_IfEqElse (params : list Exp) (e1 e2 : Exp) (n1 n2 : nat)
      (then_exp else_exp : Exp) (v : Val):
    n1 <> n2 ->
```

```
    e1 ==> (VNum n1) -> e2 ==> (VNum n2) ->
    params = [e1; e2] ->
    else_exp ==> v ->
    (If Equal params then_exp else_exp) ==> v
where "e '==>' v" := (evalExp_R e v) : type_scope.
```

Prilog 2: Potpun kôd za relaciju `evalExp_R`.

```
Example e1 := Op Add [(Const 5); (Const 2)].
Example e2 := Op Head [Const 13; Op Cons [Const 9; Const 11]].
Example e3 := Op Cons [e1; e2].
Example v1 := (VPair (VNum 7) (VNum 13)).

Theorem simple_exp : forall v, e3 ==> v -> v = v1.
Proof.
  unfold e3. unfold v1. intros. destruct v.
  - inversion H.
  - f_equal.
    + inversion H. subst. inversion H5. subst.
      inversion H3. subst. f_equal. inversion H6.
      subst. inversion H1. subst. inversion H2. reflexivity.
    + inversion H. subst. inversion H5. subst.
      inversion H4. subst. inversion H2.
Qed.
```

Prilog 3: Primer dokaza ispravnog tumačenja jednog konkretnog izraza.

```
Definition x : string := "x"%string.
Compute empty_env x. (* = None : option Val *)

Example x_init := (x |-> (VPair (VNum 1) (VNum 2))).
Compute x_init x. (* = Some (VPair (VNum 1) (VNum 2)) : option Val *)
Example x_overwr := (x !-> (VNum 3) ; x_init).
Compute x_overwr x. (* = Some (VNum 3) : option Val *)
```

Prilog 4: Primer korišćenja okruženja definisanog u kodu 3.5.

```
Tactic Notation "neg_bool_finish" tactic(c) :=
  c; [constructor | assumption | f_equal].

Lemma negating_bool_lemma : forall (b : bool) (e : Exp),
    e ==> (Bool b) ->
    (Op Sub [Const 1; e]) ==> (Bool (negb b)).
Proof.
  intros. destruct b.
  - simpl in *. destruct e.
    + (* e = Const n *)
      inversion H. subst.
      neg_bool_finish apply (E_OpSub _ (Const 1) (Const 1) 1 1).
    + (* e = Op op params *)
    neg_bool_finish apply (E_OpSub _ (Const 1) (Op op params) 1 1).
    + (* e = If test params then_exp else_exp *)
      neg_bool_finish apply (E_OpSub _ (Const 1) (If test params e4 e5) 1 1).
  - simpl in *. destruct e.
    + (* e = Const n *)
      inversion H. subst.
      neg_bool_finish apply (E_OpSub _ (Const 1) (Const 0) 1 0).
    + (* e = Op op params *)
      neg_bool_finish apply (E_OpSub _ (Const 1) (Op op params) 1 0).
    + (* e = If test params then_exp else_exp *)
      neg_bool_finish apply (E_OpSub _ (Const 1) (If test params e4 e5) 1 0).
Qed.
```

Prilog 5: Postupan dokaz leme o negiranju bulovskih izraza.

```
Proof.
  intros params e1 e2 n1 n2 n3 n4 then_exp else_exp H1 H2 Hp Ht He.
  destruct (reflect_eq n1 n2).
  - subst. rename n2 into n.
    apply (E_IfEqThen _ e1 e2 n); try assumption; try f_equal.
  - apply (E_IfEqElse _ e1 e2 n1 n2 then_exp else_exp);
      try assumption; try f_equal.
Qed.
```

Prilog 6: Dokaz leme date u kodu 4.3.

```
Lemma if_equal_lemma_b : forall (params : list Exp) (e1 e2 : Exp)
           (n1 n2 : nat) (b3 b4 : bool) (then_exp else_exp : Exp),
 params = [e1; e2] ->
 e1 ==> (VNum n1) -> e2 ==> (VNum n2) ->
 then_exp ==> (Bool b3) -> else_exp ==> (Bool b4) ->

 (If Equal params then_exp else_exp) ==> Bool (if n1 =? n2 then b3 else b4).
```

Prilog 7: Lema kojom se pokriva bulovski slučaj u granama izraza `If Equal`.

```
Compute eval empty_env
(Let TMP (Call DTS [Call EVEN [Const 12]])
         (Call PRINT [Var TMP]))
(init_state [dts_dec; print_dec; even_dec] EmptyString).

(* = (Res (VNum 0),
     {| decls :=
       [Defun 100 [77] (Op Cons [Op Add [Var 77; Const 48]; Const 0]);
        Defun 112 [83] (If Equal [Var 83; Const 0] (Const 0) (Let 84
          (Op Write [Op Head [Var 83]]) (Call 112 [Op Tail [Var 83]])));
        Defun 101 [78] (If Equal [Var 78; Const 0] (Const 1)
          (Op Sub [Const 1; Call 101 [Op Sub [Var 78; Const 1]]]))];
      input := "";
      output := "1"
     |}) : result * state  *)
```

Prilog 8: Vrednost izraza koji sadrži sve implementirane funkcije iz poglavlja 5.2.

```
(* stanje steka, dela memorije i programa *)
[] { RX := 0; RY := 0; ... } [iPushReg Y; iPushConst 1; iMov Y;
 iFunCall fEVEN; iMov AX; iMov Y; iPushReg AX]
(* nakon 3. instrukcije *)
[0] { RX := 0; RY := 1; ... } [iFunCall fEVEN; iMov AX; iMov Y; iPushReg AX]
(* nakon 4. (poziv funkcije) *)
[0] { RX := 0; RY := 1; ... } [iPushReg Y; iPushConst 0;
 iEqual; iPushConst 1; iNoopThen; iPushConst 1; iPushReg Y;
```

```
 iPushReg Y; iPushConst 1; iArith iSub; iMov Y; iFunCall fEVEN;
 iMov AX; iMov Y; iPushReg AX; iArith iSub; iNoopElse; iMov AX;
 iMov Y; iPushReg AX]
(* nakon 13. (rekurzivni poziv) *)
[1; 1; 0] { RX := 0; RY := 0; ... } [iFunCall fEVEN; iMov AX;
 iMov Y; iPushReg AX; iArith iSub; iMov AX; iMov Y; iPushReg AX]
(* nakon 16. *)
[0; 0; 1; 1; 0] { RX := 0; RY := 0; ... } [iEqual; iPushConst 1;
 iNoopThen; iPushConst 1; iPushReg Y; iPushReg Y; iPushConst 1;
 iArith iSub; iMov Y; iFunCall fEVEN; iMov AX; iMov Y; iPushReg AX;
 iArith iSub; iNoopElse; iMov AX; iMov Y; iPushReg AX; iArith iSub;
 iMov AX; iMov Y; iPushReg AX]
(* nakon 17. (izlaz iz rekurzije) *)
[1; 1; 0] { RX := 0; RY := 0; ... } [iPushConst 1; iMov AX;
 iMov Y; iPushReg AX; iArith iSub; iMov AX; iMov Y; iPushReg AX]
(* nakon 25. *)
[0] { RX := 0; RY := 0; ... } []
```

Prilog 9: Uvid u stanja tokom izvršavanja `targDenote` funkcije pozvane za program dobijen pozivom: `codegen [Call EVEN [Const 1]]`.