math.e

Hrvatski matematički elektronički časopis

Logičko programiranje

logička rezolucija **prolog**

Vedran Čačić, Petar Paradžik i Mladen Vuković

veky@math.hr , paradzik42@gmail.com , vukovic@math.hr

Sažetak

Logičko programiranje je paradigma nastala početkom sedamdesetih godina proš-log stoljeća kao direktna posljedica rada na automatiziranim dokazivačima teorema. U logičkom programiranju logika se koristi kao deklarativni jezik za opisivanje problema, a dokazivač teorema kao mehanizam za rješavanje problema. U ovom članku pokušat ćemo prvo objasniti teorijsku osnovu logičkog programiranja, a onda ćemo u posljednjem dijelu pokušati povezati teoriju s primjerima programa u programskom jeziku Prolog.

Uvod

Za razliku od imperativne i funkcijske paradigme koje se, redom, oslanjaju na Turingov stroj i λ–račun, logičko programiranje temelji se na logici prvog reda ograničenoj s takozvanim Hornovim klauzulama. Važno svojstvo Hornovih klauzula je da predstavljaju formule koje se mogu interpretirati kao procedure i time efikasno implementirati na računalu. Još jedno važno svojstvo je da logikom prvog reda, ograničenom na Hornove klauzule, možemo izraziti sve Turing–izračunljive funkcije (za definiciju vidi [10]).

U srži logičkog programiranja leži rezolucija. To je metoda kojom se dokazuje valjanost formule tako da se pokaže da negacija formule nije ispunjiva. Rezolucija u logici sudova je potpuna metoda, što znači da može zamijeniti bilo koji deduktivni sistem. Još se početkom sedamdesetih godina shvatilo da rezolucija može biti interpreter za jezik logike prvog reda ograničen s Hornovim klauzulama. Tako je rođen prvi logički programski jezik opće namjene — Prolog. Prolog sadrži samo tri osnovna konstrukta: činjenice, pravila i upite. Skup činjenica čini bazu znanja koja, zajedno s pravilima, definira logički program. Možemo kratko reći da se programiranje u Prologu svodi na konstruiranje pravila i odgovarajuće baze znanja.

Primjerima logičkih (deklarativnih) programa obično se opisuju neke stablaste strukture (primjerice, obiteljsko stablo). Na žalost, većina jednostavnijih (i po našem mišljenju, potpuno nezanimljivih) primjera u Prologu su vezani uz obiteljska stabla. No, Prolog se koristi u implementaciji kompleksnijih modela u područjima umjetne inteligencije i ekspertnih sustava. Jedan zanimljivi eksperiment je IBM Watson koji u pozadini ima „ pattern matching” algoritme implementirane u Prologu. IBM Watson je sustav znanja koji može odgovarati na pitanja koja su postavljena u prirodnom jeziku. Taj

sustav je 2011. godine pobijedio u kvizu Jeopardy! u SAD–u dvojicu tadašnjih rekordera (vidi primjerice: http://en.wikipedia.org/wiki/Watson_(computer)).

Napomenimo da je Prolog, kao i većina drugih logičkih programskih jezika, namijenjen za simbolički račun, u kojemu se naglasak stavlja na transformacije matematičkih izraza koji su zadani nizom simbola. Takve transformacije čuvaju semantičko značenje izraza, pa je svaki rezultat, koji je posljedica takvih transformacija, egzaktan. Jednu vrstu ovakvih transformacija ćemo uvesti kada budemo definirali metodu unifikacije u logici prvog reda. Ovo je, ujedno, i glavna razlika između Prologa i, primjerice, Fortran–a ili MATLAB–a, programskih jezika koji su prvenstveno namijenjeni za numerički račun, unutar kojeg su u središtu pozornosti floating–point reprezentacija (ograničena aproksimacija) realnih brojeva i floating–point aritmetika.

Članak je podijeljen u tri točke. U prvoj točki se bavimo metodom rezolucije u logici sudova, u drugoj općom rezolucijom u logici prvog reda, te u trećoj programskim jezikom Prolog. Želimo još naglasiti da je ovaj članak zapravo skraćena verzija diplomskog rada [7]. Dokaze, koji su ovdje ispušteni, možete pronaći u navedenom diplomskom radu.

1 Rezolucija u logici sudova

Logika sudova je odlučiva teorija. To znači da postoji algoritam koji za svaku formulu logike sudova u konačno mnogo koraka odlučuje je li ona valjana ili ne. Neki algoritmi odlučivanja su, primjerice, tablice istinitosti i semantička stabla (vid [9]). U ovoj točki predstavljamo još jedan algoritam odlučivanja — rezoluciju. Ovdje, na početku, želimo naglasiti da programski jezik Prolog koristi rezoluciju za logiku prvog reda. No, odlučili smo prvo predstaviti metodu rezolucije za logiku sudova kako bismo na jednoj vrlo jednostavnoj teoriji objasnili glavne ideje.

Rezolucija je jednostavno pravilo zaključivanja koje se iterativno primjenjuje na formulu u konjunktivnoj normalnoj formi, sve dok se ne dobije formula s određenim svojstvom. Kao i metoda semantičkih stabala, rezolucija je metoda opovrgavanja — da bismo dokazali da je formula valjana, dokazujemo da njena negacija nije ispunjiva.

Smatramo da je čitatelj upoznat s osnovnim pojmovima klasične logike sudova kao što su: alfabet, formula, interpretacija, istinitost formule za zadanu interpretaciju, relacija logičke posljedice, te s raznim vrstama formula (ispunjiva, valjana, oboriva i antitautologija). Svi detalji mogu se vidjeti, primjerice, u [9].

Pošto ćemo dosta koristiti pojmove vezane uz normalne forme, ovdje ćemo ponoviti njihove definicije. Literal je svaka propozicionalna varijabla ili njena negacija. Disjunkcija formula je svaka formula oblika $A_1 \vee \ldots \vee A_n$, gdje su A_i proizvoljne formule. Sasvim analogno se definira pojam konjunkcije formula. Elementarna disjunkcija je svaka disjunkcija literala. Konjunktivna normalna forma je svaka konjunkcija elementarnih disjunkcija. U daljnjem tekstu često ćemo umjesto "konjunktivna normalna forma" pisati samo kratko "KNF".

Iz teorema o normalnim formama znamo da za svaku formulu F logike sudova postoji KNF G tako da vrijedi $F \Leftrightarrow G$. Važno je naglasiti da postoji algoritam polinomne vremenske složenosti koji za svaku formulu F određuje KNF G tako da vrijedi: formula F je ispunjiva ako i samo ako je formula G ispunjiva (vidi [5]).

U daljnjem tekstu promatrat ćemo skupovni zapis KNF. Taj zapis ćemo

sada formalno definirati.

Definicija 1. *Klauzula je proizvoljan konačan skup literala. Prazna klauzula je prazan skup, i označavamo je s* $\square$. *Formula u klauzalnom obliku je proizvoljan konačan skup klauzula. Prazna formula u klauzalnom obliku je prazan skup klauzula, i označavamo je s* $\varnothing$.

Klauzula odgovara jednoj elementarnoj disjunkciji, dok klauzalni oblik formule odgovara konjunkciji elementarnih disjunkcija, odnosno, formuli u konjunktivnoj normalnoj formi. Primijetimo da su barem dvije prednosti skupovnog zapisa: nema ponavljanja istih literala u jednoj klauzuli, te ne moramo razmišljati o poretku literala i klauzula.

Od sada pa nadalje za prikaz formule u klauzalnom obliku koristit ćemo skraćenu notaciju tako što ćemo ukloniti vitičaste zagrade, a negaciju literala ćemo prikazati crticom iznad samog literala. Na primjer, formulu $\{\{p, r\}, \{\neg q, \neg p, q\}, \{p, \neg p, q\}\}$ zapisujemo kao $\{pr, \bar{q}\bar{p}q, p\bar{p}q\}$.

Formulu u klauzalnom obliku možemo dodatno pojednostavniti uklanjanjem trivijalnih klauzula koje sada definiramo.

Definicija 2. *Za dva literala kažemo da su komplementarni literali ako se jedan može dobiti negacijom drugog. Za klauzulu kažemo da je trivijalna klauzula ako sadrži barem jedan par komplementarnih literala.*

Sljedeći teorem naglašava zašto iz skupa klauzula možemo izostaviti trivijalne klauzule.

Teorem 3. *Neka je* S *skup klauzula, te neka je* $C \in S$ *trivijalna klauzula. Tada je skup klauzula* $S \setminus \{C\}$ *logički ekvivalentan sa skupom klauzula* S.

Sada definiramo pravilo rezolucije. Ako je l neki literal tada s l^{c} označavamo suprotni literal.

Definicija 4. *[Pravilo rezolucije u logici sudova] Neka su* C_1 *i* C_2 *klauzule takve da vrijedi* $l \in C_1$ *i* $l^{\mathrm{c}} \in C_2$. *Tada za klauzule* C_1 *i* C_2 *kažemo da su podudarajuće klauzule, te da se podudaraju na paru komplementarnih literala* l *i* l^{c}. *Rezolventa klauzula* C_1 *i* C_2 *je skup* $(C_1 \setminus \{l\}) \cup (C_2 \setminus \{l^{\mathrm{c}}\})$. *Rezolventu klauzula* C_1 *i* C_2 *označavamo s* $Res(C_1, C_2)$. *Klauzule* C_1 *i* C_2 *nazivamo roditelji klauzule* $Res(C_1, C_2)$.

Primjer 5. *Par klauzula* $C_1 = \{ab\bar{c}\}$ *i* $C_2 = \{bc\bar{e}\}$ *podudara se na paru komplementarnih literala:* c *i* $\bar{c}$. *Rezolventa klauzula* C_1 *i* C_2 *je:*

$$Res(C_1, C_2) = (\{ab\bar{c}\} \setminus \{\bar{c}\}) \cup (\{bc\bar{e}\} \setminus \{c\}) = \{ab\} \cup \{b\bar{e}\} = \{ab\bar{e}\}.$$

Ako se dvije klauzule podudaraju na više od jednom paru komplemantarnih literala, tada je očito njihova rezolventa trivijalna. Zatim, lako je vidjeti da pravilo rezolucije čuva ispunjivost. Točnije, vrijedi sljedeći teorem.

Teorem 6. *Neka je* C *rezolventa klauzula* C_1 *i* C_2. *Tada vrijedi: rezolventa* C *je ispunjiva ako i samo ako su klauzule* C_1 *i* C_2 *ispunjive.*

Koristeći pravilo rezolucije sada navodimo metodu rezolucije za logiku sudova. Metodu ćemo navesti u formi algoritma. Ulazni podatak tog algortima je neki skup klauzula S, a izlaz algoritma je jedna od sljedećih poruka: *skup*S je ispunjiv}, odnosno *skup*S nije ispunjiv}.

Metoda rezolucije u logici sudova:
Ulaz: Skup klauzula S.
Neka je $S_0 = S$.

Ponavljaj sljedeće korake:

(1)

Odaberi par podudarajućih klauzula $\{C_1,\ C_2\} \subseteq S_i$ koji do sada nije bio odabran. Ako takav par ne postoji, postupak se zaustavlja s porukom: *skup*S je ispunjiv.}

(2)

Odredi $C = Res(C_1, C_2)$ koristeći pravilo rezolucije.

(3)

Ako je $C = \square$, postupak staje s porukom: *skup*S nije ispunjiv.} Inače, ako C nije trivijalna klauzula tada je $S_{i+1} = S_i \cup \{C\}$, a ako je C trivijalna klauzula tada je $S_{i+1} = S_i$. Postupak se nastavlja vraćanjem na korak 1.

Dakle, metoda rezolucije se sastoji od iterativne primjene pravila rezolucije na skup klauzula s ciljem da se dobije prazna klauzula $\square$. Naime, znamo da pravilo rezolucije čuva ispunjivost, stoga, ako u jednom trenutku dobijemo rezolventu koja je prazna klauzula (sjetimo se da prazna klauzula nije ispunjiva), možemo zaključiti da početni skup klauzula S nije ispunjiv.

Metoda rezolucije se zaustavlja ako pronađemo rezolventu koja je prazna klauzula ili ako nam ponestane parova klauzula pomoću kojih možemo generirati nove rezolvente. Budući da imamo samo konačan broj različitih klauzula u početno zadanom skupu klauzula, zaključujemo da se metoda rezolucija zaustavlja za proizvoljan ulaz.

Sada nam je cilj istaknuti najvažnija svojstva metode rezolucije. To su teoremi adekvatnosti i potpunosti. Kako bismo točno mogli formulirati navedene teoreme, moramo definirati pojam potpunog, odnosno nepotpunog rezolucijskog stabla.

Metodu rezolucije možemo zapisati u obliku tzv. rezolucijskog stabla. Listovi stabla označeni su klauzulama iz S koje sudjeluju u metodi rezolucije, dok su unutarnji čvorovi označeni rezolventama. Kažemo da je takvo stablo dobiveno iz skupa klauzula S, odnosno da je to rezolucijsko stablo za S. Korijen stabla može biti označen bilo kojom rezolventom koja se može dobiti u nekom koraku metode rezolucije. To znači da u svakom koraku metode rezolucije možemo promatrati novo rezolucijsko stablo. Jasno, nas će zanimati samo ona stabla kojima je korijen označen praznom klauzulom. Takvo stablo ćemo nazivati potpuno rezolucijsko stablo. Inače ćemo reći da se radi o nepotpunom rezolucijskom stablu.

Teorem 7. *[Adekvatnost metode rezolucije za logiku sudova]*
Neka je S skup klauzula logike sudova. Ako postoji potpuno rezolucijsko stablo za skup klauzula S tada skup klauzula S nije ispunjiv.

Teorem 8. *[Potpunost metode rezolucije za logiku sudova]*
Ako skup klauzula S nije ispunjiv tada postoji potpuno rezolucijsko stablo za skup klauzula S.

Vremenska složenost metode rezolucije nije tako očita kao kod, primjerice, tablica istinitosti. Ona se može mjeriti brojem različitih rezolventi koje se generiraju tijekom zaključivanja. Haken je 1985. godine dokazao da postoji niz valjanih formula $F_1, F_2, \ldots$, takvih da opovrgavanje svake formule F_i zahtijeva generiranje barem c^n rezolventi, gdje je $c > 1$. Svaka od formula F_i izražava Dirichletov princip: ako $i+1$ predmet bilo kako rasporedimo u i kutija, tada barem jedna kutija sadrži barem dva predmeta. Dokaz se može naći u [2]. Dakle, metoda rezolucije ima eksponencijalnu vremensku složenost. Ovo je očekivano, jer je poznato da je problem valjanosti

co–NP–potpun.

Jedna od primjena rezolucije u logici sudova je u Davis–Putnam–Logemann–Loveland algoritmu za rješavanje NP–potpunog problema ispunjivosti logičke formule poznatog kao problem SAT. Osim što ima povijesni značaj time što je bio prvi primjer NP–potpunog problema, problem SAT je ujedno i jedan od najistraživanijih problema u praksi. Moderne varijante Davis–Putnam–Logemann–Lovelandovog algoritma, starog više od pola stoljeća, i danas se koriste za njegovo rješavanje (vidi, primjerice, [5] i [6]).

2 Rezolucija u logici prvog reda

Kao i rezolucija u logici sudova, rezolucija u logici prvog reda (naziva se i opća rezolucija) je, također, adekvatna i potpuna na skupu svih formula logike prvog reda. Međutim, ispostavlja se da takva rezolucija nije efikasna za implementaciju na računalu. U tu svrhu promatraju se razne restrikcije opće rezolucije. U ovoj točki bavimo se jednom od takvih restrikcija, i to onom koja je početkom sedamdesetih godina prošlog stoljeća rezultirala pojavom prvog logičkog programskog jezika Prolog. Riječ je o SLD–rezoluciji (eng. Selective Linear Definite clause resolution) kojom ispitujemo ispunjivost samo onih formula logike prvog reda koje se mogu izraziti pomoću Hornovih klauzula.

Hornove klauzule imaju svojstvo da se sastoje od najviše jednog pozitivnog literala, tj. literala bez negacije. Takve klauzule, osim deklarativne imaju i proceduralnu interpretaciju, odnosno, mogu se promatrati kao procedure i time efikasno implementirati na računalu. Osim toga, Hornove klauzule imaju izražajnu moć Turingova stroja, što znači da pomoću njih možemo izraziti sve Turing-izračunljive funkcije. SLD–rezoluciju zapravo možemo promatrati kao interpreter programskog jezika kojeg čini skup Hornovih klauzula.

Danas svaka varijanta rezolucije u logici prvog reda, pa tako i SLD–rezolucija, uključuje moćnu metodu unifikacije. Metoda unifikacije rješava problem podudaranja terma[1] — za dana dva terma koji sadrže neke individualne varijable, treba pronaći, ako postoji, najjednostavniju supstituciju individualnih varijabli termima, takvu da nakon nje početna dva terma budu jednaka. Ovdje ćemo opisati takozvanu Martelli-Montanarijevu unifikaciju iz 1982. godine.

Za razliku od opće rezolucije, SLD–rezolucija nije potpuna na skupu svih formula logike prvog reda. Razlog je taj što Hornovim klauzulama ne možemo izraziti sve valjane formule logike prvog reda. S druge strane, SLD–rezolucija je potpuna kad se ograniči samo na Hornove klauzule. To znači da za svaku neispunjivu Hornovu klauzulu postoji izvod SLD–rezolucijom koji je prazna klauzula.

Dok je rezolucija u logici sudova imala svojstvo da staje s radom u konačno mnogo koraka za svaki ulaz, SLD-rezolucija (a time i opća rezolucija) nema to svojstvo. SLD–rezolucija može raditi vječno ako joj na ulaz damo formulu koja je ispunjiva. Ovo je posljedica činjenice da logika prvog reda nije odlučiva teorija (vidi [10]).

2.1 Skolemova normalna forma

Na početku moramo odrediti oblik formula s kojima ćemo raditi. U uvodu smo spomenuli da će metoda unifikacije raditi određene supstitucije na formulama. To znači da je poželjno da se formule sastoje od univerzalnih kvantifikatora. Oni nam garantiraju da ćemo individualne varijable u formuli moći zamijeniti proizvoljnim termima. Smatramo da je čitatelj upoznat s pojmom preneksne normalne forme (definicija i primjeri se mogu vidjeti, primjerice, u [9]).

U ovom dijelu promatramo formule u preneksnoj konjunktivnoj normalnoj formi, i to one koje se sastoje samo od univerzalnih kvantifikatora. Individualne varijable ćemo općenito označavati s $x, y, z \ldots$, a konstantske simbole s $a, b, c \ldots$.

Definicija 9. *Neka je $F \equiv \forall x_1 \cdots \forall x_n F'$ zatvorena formula u preneksnoj konjuktivnoj normalnoj formi gdje je F' otvorena formula. Skolemova normalna forma formule F sastoji se od formule F' zapisane kao skup klauzula.*

Primjer 10. *Neka je $F \equiv \forall x \forall z([p(f(x)) \vee \neg p(g(y)) \vee q(y)] \wedge [\neg q(y) \vee \neg p(g(y)) \vee q(x)])$ neka zatvorena formula u preneksnoj konjunktivnoj normalnoj formi. Skolemovu normalnu formu formule F predstavlja skup klauzula $\{\{p(f(x)), \neg p(g(y)), q(y)\}, \{\neg q(y), \neg p(g(y)), q(x)\}\}$.*

Za proizvoljnu formulu logike prvog reda želimo naći njezinu Skolemovu normalnu formu. Dok smo u logici sudova imali logičku ekvivalenciju između formule i njezinog klauzalnog oblika (odnosno konjunktivne normalne formule), u logici prvog reda očuvana je samo ispunjivost. O ovome govori Skolemov teorem, čiji dokaz možete vidjeti, primjerice, u [9].

Teorem 11. *[T. Skolem] Neka je F zatvorena formula. Tada postoji formula u Skolemovoj normalnoj formi F' takva da vrijedi sljedeće:*

$$F \text{ je ispunjiva ako i samo ako je } F' \text{ ispunjiva.}$$

Skolemov teorem na neki način govori da se formula u logici prvog reda može svesti na formulu logike sudova eliminacijom egzistencijalnih kvantifikatora i promatranjem samo Skolemove normalne forme. Upravo je eliminacija egzistencijalnih kvantifikatora razlog zašto se gubi logička ekvivalencija između formule F i njezine Skolemove normalne forme F'.

2.2 Unifikacija

Prije opisa metode unifikacije uvedimo nekoliko definicija. Cilj nam je formalno definirati pojam instanciranja individualne varijable termom.

Definicija 12. *Neka su $x_1, \ldots, x_n$ individualne varijable, a $t_1, \ldots, t_n$ termi. Supstitucija terma za individualne varijable je skup $\{x_1 \leftarrow t_1, \ldots, x_n \leftarrow t_n\}$, pri čemu za sve $i, j \in \{1, \ldots, n\}, i \neq j$, vrijedi $x_i \neq x_j, x_i \neq t_i$. Prazna supstitucija je prazan skup.*

Supstitucije ćemo označavati s λ, μ, σ, θ, dok ćemo praznu supstituciju označavati s ϵ. Supstitucije želimo provoditi na svim formulama.

Definicija 13. *Izraz je term, literal, klauzula ili skup klauzula. Neka je E neki izraz i $\theta = \{x_1 \leftarrow t_1, \ldots, x_n \leftarrow t_n\}$ neka supstitucija. Instancu izraza E, u oznaci $E\theta$, definiramo kao simultanu supstituciju pojave svake individualne varijable x_i u izrazu E termom t_i.*

Primjer 14. *Klauzula $E = \{p(x), q(f(y))\}$ predstavlja jedan izraz, a $\theta = \{x \leftarrow y, y \leftarrow f(a)\}$ jednu supstituciju. Instanca izraza E sa supstitucijom θ je $E\theta = \{p(y), q(f(f(a)))\}$. U definiciji smo naglasili da se mora raditi o simultanoj supstituciji, što znači da ne smijemo prvo supstituirati x s y da dobijemo $\{p(y), q(f(y))\}$, a tek onda supstituirati y s $f(a)$ da dobijemo $\{p(f(a)), q(f(f(a)))\}$.*

Rezultat supstitucije očito ne mora biti izraz bez individualnih varijabli. Supstitucija može samo „preimenovati" individualne varijable. Takva je, primjerice, sljedeća supstitucija: $\{x \leftarrow y, z \leftarrow w\}$.

Definicija 15. *Neka su* $\theta = \{x_1 \leftarrow t_1, \ldots, x_n \leftarrow t_n\}$ *i* $\sigma = \{y_1 \leftarrow s_1, \ldots, y_m \leftarrow s_m\}$ *dvije supstitucije, te* $X = \{x_1, \ldots, x_n\}$ *i* $Y = \{y_1, \ldots, y_m\}$ *skupovi individualnih varijabli koji se redom pojavljuju u supstitucijama* θ *i* σ*. Kompoziciju supstitucija* θ *i* σ*, u oznaci* $\theta\sigma$*, definiramo kao supstituciju:*

$$\theta\sigma = (\bigcup_{i=i\ x_i \in X\ x_i \neq t_i\sigma}^{n} \{x_i \leftarrow t_i\sigma\}) \cup (\bigcup_{j=1\ y_j \in Y\ y_j \notin X}^{m} \{y_j \leftarrow s_j\}).$$

Primjer 16. *Neka su* $\theta = \{x \leftarrow f(y), y \leftarrow f(a), z \leftarrow u\}$ *i* $\sigma = \{y \leftarrow g(a), u \leftarrow z, v \leftarrow f(f(a))\}$ *dvije supstitucije, te* $X = \{x, y, u, z\}$ *i* $Y = \{y, u, v, z\}$ *skupovi individualnih varijabli.Tada imamo:* $\theta\sigma = \{x \leftarrow f(g(a)), y \leftarrow f(a), u \leftarrow z, v \leftarrow f(f(a))\}$.

Primijetimo da se supstitucija $z \leftarrow z = (z \leftarrow u)\sigma$ *ne pojavljuje u skupu* $\theta\sigma$*. Supstitucija* $y \leftarrow g(a)$ *iz* σ *se, također, ne pojavljuje u skupu* $\theta\sigma$*, jer je* $y \in X$*.*

Sada definiramo jednu posebnu vrstu supstitucije koju ćemo koristiti u metodi unifikacije.

Definicija 17. *Neka je* $U = \{p_1, \ldots, p_n\}$ *skup atomarnih formula. Unifikator za* U *je supstitucija* θ *takva da vrijedi* $p_1\theta = \cdots = p_n\theta$*. U tom slučaju ćemo reći da atomarne formule* p_i*,* $i \in \{1, \ldots, n\}$ *možemo unificirati. Najopćenitiji unifikator za* U *je unifikator* μ *takav da za svaki unifikator* θ *od* U *postoji supstitucija* λ *takva da vrijedi* $\theta = \mu\lambda$*.*

Primjer 18. *Supstitucije* $\theta_1 = \{x \leftarrow f(a), y \leftarrow f(g(a)), z \leftarrow f(g(a))\}$, $\theta_2 = \{x \leftarrow f(a), y \leftarrow a, z \leftarrow a\}$ *i* $\mu = \{x \leftarrow f(a), z \leftarrow y\}$ *su unifikatori za skup atomarnih formula* $\{p(f(x), g(y)), p(f(f(a)), g(z))\}$*. Supstitucija* μ *je i najopćenitiji unifikator, pa supstitucije* θ_1 *i* θ_2 *možemo izraziti kao* $\theta_1 = \mu\{y \leftarrow f(g(a))\}$ *i* $\theta_2 = \mu\{y \leftarrow a\}$*.*

Očito ne možemo sve atomarne formule unificirati. To su, primjerice, one kojima se relacijski, odnosno, funkcijski simboli razlikuju. Također, ne možemo unificirati atomarne formule oblika $p(x)$ i $p(f(x))$, budući da bilo koja supstitucija mora simultano supstituirati terme x i $f(x)$.

Primjerice, problem unifikacije skupa atomarnih formula $\{p(f(x), g(y)), p(f(f(a)), g(z))\}$ svodi se na ispitivanje sljedeće "jednakosti": $p(f(x), g(y)) = p(f(f(a)), g(z))$. Očito je da se dvije atomarne formule mogu unificirati ako i samo ako se sastoje od istih relacijskih simbola i primaju jednak broj argumenata. Iz tog razloga govorit ćemo o skupu jednakosti terma za skup atomarnih formula, Primjerice, skup jednakosti terma za skup atomarnih formula $\{p(f(x), g(y)), p(f(f(a)), g(z))\}$ je sljedeći skup: $\{f(x) = f(f(a)), g(y) = g(z)\}$.

Metoda unifikacije u svakoj iteraciji pokušava primijeniti određenu \textbf {transformaciju} na skup jednakosti terma. Zanimaju nas samo one transformacije koje čuvaju skup svih unifikatora, jer upravo te transformacije čuvaju i najopćenitiji unifikator. Definirajmo sada dvije takve transformacije:

-

Redukcija terma. Neka je $f(t_1, \ldots, t_n) = f(s_1, \ldots, s_n)$ jednakost terma, gdje su t_i, s_i, za $i \in \{1, \ldots, n\}$, termi, a f je funkcijski

simbol. Transformacija ovu jednakost zamjenjuje sa sljedećim skupom jednakosti terma: $t_1 = s_1, t_2 = s_2, \ldots, t_n = s_n$. Ako je $n = 0$, tada je f konstantski simbol pa jednakost jednostavno brišemo.

- Eliminacija varijable. Neka je $x = t$ jednakost terma, gdje je x individualna varijabla, a t proizvoljan term. Transformacija tada zamjenjuje svaku pojavu individualne varijable x termom t u svakoj jednakosti iz skupa jednakosti terma (pritom se jednakost $x = t$ ne briše iz skupa).

Sada navodimo metodu unifikacije. Zapisujemo je u obliku algoritma.

Metoda unifikacije u logici prvog reda:
Ulaz: Skup jednakosti terma.
Izvršavaj sljedeće korake (1)–(4) dok god se jedan od njih može primijeniti. Ako u jednom trenutku dobijemo skup jednakosti terma sa svojstvom da su sve jednakosti oblika $x = t$, gdje je x individualna varijabla koja se ne pojavljuje više nigdje na lijevoj strani neke jednakosti, postupak se zaustavlja s porukom: skup jednakosti terma na izlazu je najopćenitiji unifikator za zadani skup jednakosti terma.

(1)

Odaberi bilo koju jednakost oblika $t = x$, gdje je x individualna varijabla, a t term koji nije individualna varijabla, i transformiraj je u jednakost $x = t$.

(2)

Odaberi bilo koju jednakost oblika $x = x$ i izbaci je iz skupa jednakosti.

(3)

Odaberi bilo koju jednakost oblika $t' = t''$, gdje t' i t'' nisu individualne varijable. Ako početni funkcijski simboli od t' i t'' nisu jednaki, postupak se zaustavlja s porukom: skup jednakosti terma se ne može unificirati. Inače primijeni transformaciju redukcije terma na jednakost $t' = t''$.

(4)

Odaberi bilo koju jednakost oblika $x = t$, gdje je x individualna varijabla koja se pojavljuje još barem na jednom mjestu u skupu jednakosti i za koju vrijedi $x \neq t$. Ako se x pojavljuje u termu t, postupak se zaustavlja s porukom: skup jednakosti terma se ne može unificirati. Inače primijeni transformaciju eliminacije varijable koristeći jednakost $x = t$.

2.3 Hornove klauzule

Postoji nekoliko razloga za uvođenje Hornovih klauzula. Prvi je činjenica da se Hornovim klauzulama mogu izraziti sve parcijalno rekurzivne funkcije (vidi [3]), a time i sve Turing–izračunljive funkcije. Drugi razlog je taj što se Hornove klauzule mogu interpretirati kao jednostavne procedure. Ako uzmemo u obzir još i činjenicu da je rezolventa Hornovih klauzula i dalje Hornova klauzula, iste postaju prikladne za implementaciju na računalu. Treći razlog tiče se činjenice da su Hornove klauzule jedna od najboljih restrikcija logike prvog reda s ovim svojstvima. Počnimo s definicijom Hornovih klauzula.

Definicija 19. *Hornova klauzula je klauzula s najviše jednim pozitivnim literalom. Ako je* $\{A, \neg B_1, \ldots, \neg B_n\}$ *Hornova klauzula (*$A, B_1, \ldots, B_n$ *su atomarne formule) tada tu klauzulu zapisujemo i u sljedećem obliku:* $A \leftarrow B_1, \ldots, B_n$*. Pozitivni literal* A *nazivamo glava, a negativne literale* B_i, $i \in \{1, \ldots, n\}$, *nazivamo tijelo Hornove klauzule.*

Ako Hornova klauzula sadrži samo jedan pozitivan literal A *tada je nazivamo i činjenica, te je označavamo s* $A \leftarrow$*. Ciljna klauzula je Hornova klauzula bez glave i označavamo je s* $\leftarrow B_1, \ldots, B_n$*. Programska klauzula je Hornova klauzula s glavom i tijelom koje se sastoji od barem jednog negativnog literala.*

Ovdje koristimo simbol „ $\leftarrow$" kao oznaku za implikaciju, jer osim što formula ima deklarativnu interpretaciju (da bismo dokazali A, moramo dokazati $B_1, \ldots, B_n$), formulu možemo interpretirati kao proceduru koja se sastoji od niza naredbi: da bismo izračunali A, moramo izračunati $B_1, \ldots, B_n$. Ovo opravdava sljedeću definiciju u kojoj uvodimo osnovnu terminologiju koja se koristi u logičkom programiranju.

Definicija 20. *Skup programskih klauzula kojima su glave označene istim relacijskim simbolom nazivamo procedura. Skup procedura nazivamo (logički) program. Proceduru koja se sastoji samo od činjenica koje se dalje sastoje samo od funkcijskih i konstantskih simbola nazivamo baza znanja.*

Dakle, program se sastoji od procedura i baza znanja. Programu postavljamo upite u obliku ciljnih klauzula, te ako program stane u konačno mnogo koraka, tada odgovor na upit predstavljaju supstitucije nastale unifikacijom ciljnih i programskih klauzula (odnosno procedura i baza znanja). Rad programa ćemo opisati SLD–rezolucijom. Definirajmo sada neke posebne vrste supstitucija koje će nam trebati.

Definicija 21. *Neka je* P *neki program i* G *neka ciljna klauzula. Supstitucija* θ *za individualne varijable u ciljnoj klauzuli* G *je supstitucija korektnog odgovora ako vrijedi* $P \Rightarrow \forall(\neg G\theta)$*, odnosno, ako ciljna klauzula* $\forall(\neg G\theta)$ *logički slijedi iz programa* P*, gdje* $\forall$ *definira univerzalno zatvorenje na svim slobodnim individualnim varijablama u formuli* $\neg G\theta$*.*

Neka je dan program P, ciljna klauzula $G \equiv \neg G_1 \vee \cdots \vee \neg G_n$ i supstitucija korektnog odgovora θ. Prema definiciji vrijedi $P \Rightarrow \forall(\neg G\theta)$, odnosno, $P \Rightarrow \forall((G_1 \wedge \cdots \wedge G_n)\theta)$. Dakle, za bilo koju supstituciju σ, svaki model za program P ujedno je i model za formulu $(G_1 \wedge \cdots \wedge G_n)\theta\sigma$. Supstitucija $\theta\sigma$ će predstavljati odgovor na upit, pa ćemo pomoću nje moći iskazati teoreme adekvatnosti i potpunosti ograničene sada samo na program.

2.4 SLD–rezolucija

Kao što smo bili već napomenuli, SLD–rezolucija je restrikcija opće rezolucije na Hornove klauzule, odnosno, na program, kao što ćemo vidjeti. Ime joj je izvedeno iz SL–rezolucije (eng. Selecitive Linear clause resolution), koja je adekvatna i potpuna na skupu svih formula logike prvog reda. Sada metodu SLD–rezolucije zadajemo u obliku algoritma.

SLD–rezolucija u logici prvog reda:
Ulaz: Program P i ciljna klauzula G.
Neka je $G_0 = G$.
Ciljnu klauzulu — rezolventu G_{i+1}, izvodimo iz klauzula G_i i $C_i \in P$ odabirom literala $A_i^j \in G_i$ koji se može unificirati s glavom klauzule

C_i pomoću najopćenitijeg unifikatora σ_i:

$$\begin{aligned} G_i &\equiv \leftarrow A_i^1, \ldots, A_i^{j-1}, A_i^j, A_i^{j+1}, \ldots, A_i^{n_i}, \\ C_i &\equiv B_i^0 \leftarrow B_i^1, \ldots, B_i^{k_i}, \\ A_i^j\theta_i &\equiv B_i^0\theta_i, \\ G_{i+1} &\equiv \leftarrow (A_i^1, \ldots, A_i^{j-1}, B_i^1, \ldots, B_i^{k_i}, A_i^{j+1}, \ldots, A_i^{n_i})\theta_i. \end{aligned}$$

Definicija 22. *Izvod pomoću SLD–rezolucije, ili kratko SLD–izvod, klauzule S iz programa P je niz ciljnih klauzula $G_0, \ldots, G_n$ dobivenih tijekom rada metode SLD-rezolucije programom P i ciljnom klauzulom G na ulazu, takav da vrijedi $G_n \equiv S$. SLD–opovrgavanje klauzule G iz skupa P je SLD–izvod prazne klauzule $\square$. Pravilo kojim odabiremo literale $A_i^j \in G_i$ nazivamo pravilo izračunavanja. Pravilo kojim odabiremo klauzulu $C_i \in P$ nazivamo pravilo pretraživanja.*

Metoda SLD–rezolucije očito ne staje s radom za svaki program P i ciljnu klauzulu G. Primjer takvog programa i ciljne klauzule predstavljaju sljedeće formule: $p(x,z) \leftarrow p(x,y) \wedge p(y,z),\ \leftarrow p(a,b)$. Metoda je, također, nedeterministična u slučaju da ne definiramo pravila izračunavanja i pretraživanja.

Objasnimo sada na što se odnose riječi selective, linear i definite u nazivu SLD–rezolucije. Riječ „ selective" se odnosi na postojanje pravila izračunavanja. Jedno pravilo izračunavanja je, primjerice, pravilo LIFO („ last in, first out"). To pravilo izračunavanja za unifikaciju odabire literal koji je zadnji uveden u ciljnu klauzulu. Riječ „ linear" se odnosi na činjenicu da rezolucija generira niz ciljnih klauzula $G_0, G_1, \ldots$, takvih da se u i–tom koraku, ciljna klauzula G_{i-1} odabire za podudaranje. Riječ „ definite" se odnosi na činjenicu da se rezolucija provodi samo nad Hornovim klauzulama.

Sada iskazujemo teoreme adekvatnosti i potpunosti za SLD–rezoluciju. Ponovno napomenimo da SLD–rezolucija nije potpuna na skupu svih formula logike prvog reda, nego samo na skupu Hornovih klauzula, odnosno programu. Dokazi teorema mogu se pronaći, primjerice, u [3].

Teorem 23. *[Adekvatnost SLD–rezolucije] Neka je P skup programskih klauzula, R pravilo izračunavanja i G ciljna klauzula. Pretpostavimo da postoji SLD-opovrgavanje klauzule G iz skupa P. Neka je $\theta = \theta_1 \cdots \theta_n$ niz unifikatora koji su se koristili pri izvodu i neka je σ restrikcija od θ na individualne varijable iz G. Tada je σ supstitucija korektnog odgovora za G.*

Teorem 24. *[Potpunost SLD-rezolucije] Neka je P skup programskih klauzula, R pravilo izračunavanja, G ciljna klauzula i σ supstitucija korektnog odgovora za G. Tada postoji SLD–opovrgavanje klauzule G iz skupa P, takvo da je σ restrikcija niza unifikatora $\theta = \theta_1 \cdots \theta_n$ na individualne varijable iz G.*

Primijetimo da teorem potpunosti kaže da postoji neki izvod prazne klauzule neovisno o korištenom pravilu izračunavanja. Pravilo pretraživanja, s druge strane, određuje hoće li se izvod pronaći i koliko efikasno će pretraživanje biti. Ova pravila su važna jer definiraju proceduralni aspekt logičkog programa. Posljedica toga je da možemo imati logičke programe koji imaju istu semantičku, ali različite proceduralne interpretacije. To znači da izvod prazne klauzule može postojati, ali da SLD–rezolucija nikada ne stane s radom!

Postoji suptilna razlika između notacije koja se koristi u Prologu i notacije koju koristimo u logici prvog reda. Simbol „ $\leftarrow$" za

3 Programski jezik Prolog

kondicional zamjenjujemo simbolom „ :-", dok umjesto simbola „∧" za konjunkciju koristimo zarez. Individualne varijable pišemo velikim početnim slovom ili simbolom „ _", dok relacijske, funkcijske i konstantske simbole označavamo proizvoljnim riječima koje počinju malim početnim slovom. Negaciju označavamo simbolom „ \+". U Prologu je pravilo pretraživanja definirano tako da se potraga za programskom klauzulom (procedurom) provodi od početka (vrha) programa prema kraju (dnu) programa, dok je pravilo izračunavanja definirano tako da se za podudaranje odabire prvi literal s lijeva u ciljnoj klauzuli. Dakle, za pravilo izračunavanja Prolog koristi depth-first search (DFS) algoritam.

Sada nam je cilj na nekoliko jednostavnih primjera povezati teoriju s programima pisanim u Prologu. Primjeri koji slijede pisani su u GNU Prolog implementaciji koja se može preuzeti na adresi: http://www.gprolog.org .

3.1 Uvod i osnovni primjer

Prvo dajemo jedan vrlo jednostavan primjer kojim želimo opisati utjecanje nekih rijeka u more. Njime želimo istaknuti osnovnu sintaksu Prologa, kao i način na koji Prolog interpreter (koji implementira rezoluciju) dolazi do zaljučaka.

Primjer 25. *[Primjer programa u Prologu]*

```
utječeU(drava, sava).
utječeU(sava, dunav).
utječeU(dunav, 'crno more').

more('crno more').

uMore(C, D) :- utječeU(C, D), more(D).
uMore(C, M) :- utječeU(C, D), uMore(D, M).
```

Ovo je primjer jednog logičkog programa koji se sastoji od tri procedure, od kojih su procedure s relacijskim simbolima „ `utječeU`" i „ `more`" ujedno i baze znanja (sastoje se od činjenica koje ne sadrže individualne varijable). Ovaj vrlo jednostavan primjer iz osnovnih upita oblika „ kamo utječe neka rijeka" izvodi zaključke oblika „ voda neke rijeke završava u nekom moru", ako se „ more" definira kao „ ono što nikamo ne utječe". Činjenicu da rijeka Drava utječe u rijeku Savu u Prologu iskazujemo kao „ `utječeU(drava, sava)`". Također, činjenicu da je Crno more neko more, u Prologu iskazujemo kao „ `more('crno more')`". S druge strane, da bismo opisali da neka rijeka utječe u Crno more, ona mora ili utjecati u njega, što iskazujemo pomoću procedure „ `uMore(C, D) :- utječeU(C, D), more(D)`", ili utjecati u neku rijeku koja dalje utječe u Crno more, što iskazujemo pomoću rekurzivne procedure „ `uMore(C, M) :- utječeU(C, D), uMore(D, M)`". Efektivno, ovo je način na koji se može modelirati tranzitivno zatvorenje neke relacije.

Da bismo dokazali da Dunav utječe u Crno more, Prologu postavljamo upit pomoću ciljne klauzule $G_0 \equiv$ `:- uMore(dunav, 'crno more').` Nakon toga, Prolog pretražuje program od vrha prema dnu (prema pravilu pretraživanja) pokušavajući pritom pronaći programsku klauzulu čiju glavu može unificirati s prvim lijevim literalom u ciljnoj klauzuli G_0 (prema pravilu izračunavanja), što je literal $A_0 \equiv$ `uMore(dunav, 'crno more').` Tražena klauzula je $C_0 \equiv$ `uMore(C, D) :- utječeU(C, D), more(D),` čiju glavu

ćemo označiti s B_0. Dakle, iz unifikacije $A_0\theta_0 \equiv B_0\theta_0$ dobivamo unifikator $\theta = \{C \leftarrow \texttt{dunav}, D \leftarrow \texttt{'crno more'}\}$ i novu ciljnu klauzulu
$G_1 \equiv$ `:- utječeU(dunav, 'crno more'), more('crno more').`
Sada istim postupkom ponovno tražimo programsku klauzulu čiju glavu možemo unificirati s prvim lijevim literalom u G_1 i tako dobivamo ciljnu klauzulu $G_2 \equiv$ `:- more('crno more').` Budući da nam je ostao jedan literal u ciljnoj klauzuli G_2, kojeg možemo unificirati s glavom jedne od programskih klauzula, za rezultat dobivamo praznu klauzulu, odnosno $G_3 \equiv \square$. Time smo dobili SLD–izvod prazne klauzule i dokazali da rijeka Dunav utječe u Crno more.

```
| ?- uMore(dunav, 'crno more').
true ?
yes
```

Prologov odgovor "true" jest jedna (prazna) unifikacija koja čini ovu klauzulu istinitom. Upitnik znači da je Prolog stao čim je našao jedno rješenje, te nas pita za dalji postupak. Ako samo pritisnemo Enter, dobijemo odgovor "`yes`" kao gore. Ako pritisnemo tipku `` `;' `` (sljedeće rješenje) ili `` `a' `` (sva rješenja), dobijemo odgovor "`no`", što znači da nema više rješenja.

Pretpostavimo sada da želimo doznati koja rijeka utječe u Crno more. Taj upit izražavamo pomoću ciljne klauzule $G \equiv$ `:- uMore(X, 'crno more'),` gdje je `X` sada individualna varijabla.

Rad Prologa prikazujemo sljedećim koracima. Prvi lijevi literal u ciljnoj klauzuli nećemo posebno naglašavati.

$G_0 \equiv$ `:- uMore(X, 'crno more');`
$C_0 \equiv$ `uMore(C, D) :- utječeU(C, D), more(D),`
$\theta_0 = \{$`C` $\leftarrow$ `X`, `D` $\leftarrow$ `'crno more'`$\}$;
$G_1 \equiv$ `:- utječeU(X, 'crno more'), more('crno more');`
$C_1 \equiv$ `utječeU(dunav, 'crno more'),`
$\theta_1 = \{$`X` $\leftarrow$ `dunav`$\}$;
$G_2 \equiv$ `:- more('crno more');`
$C_2 \equiv$ `more('crno more');`
$\theta_2 = \epsilon$;
$G_3 \equiv \square$;

Iz teorema adekvatnosti slijedi da je supstitucija $\theta = \theta_0\theta_1\theta_2\restriction_{\texttt{X}}$ (gdje s $\restriction_{\texttt{X}}$ označavamo restrikciju na individualnu varijablu `X`) supstitucija korektnog odgovora i vrijedi $P \Rightarrow \neg G\theta$, odnosno $P \Rightarrow$ `uMore(dunav, 'crno more')`, gdje je P oznaka za naš program. Prolog bi nam, za ovaj upit, uzvratio odgovorom „`X = dunav`". Međutim, ovo očito nije jedino rješenje. U slučaju da želimo još jedno rješenje, Prolog se tada mora vratiti do trenutka kada je mogao drukčije supstituirati individualnu varijablu `X`. To je očito trenutak prije odabira klauzule C_0. Promotrimo rad Prologa u tom slučaju.

$G_0 \equiv$ `:- uMore(X, 'crno more');`
$C_0 \equiv$ `uMore(C, M) :- utječeU(C, D), uMore(D, M),`
$\theta_0 = \{$`C` $\leftarrow$ `X`, `M` $\leftarrow$ `'crno more'`$\}$;
$G_1 \equiv$ `:- utječeU(X, D), uMore(D, 'crno more');`
$C_1 \equiv$ `utječeU(drava, sava),`
$\theta_1 = \{$`X` $\leftarrow$ `drava`, `D` $\leftarrow$ `sava`$\}$;
$G_2 \equiv$ `:- uMore(sava, 'crno more');`
$C_2' \equiv$ `uMore(C, D) :- utječeU(C, D), more(D),`
$\theta_2' = \{$`C` $\leftarrow$ `sava`, `D` $\leftarrow$ `'crno more'`$\}$;

$G_3' \equiv$ `:- utječeU(sava, 'crno more'), more('crno more');`

Budući da rijeka Sava ne utječe direktno u Crno more, Prolog pomoću ciljne klauzule G_3' ne može generirati praznu klauzulu. U tom slučaju, Prolog se vraća do zadnjeg trenutka kad je mogao odabrati drugu programsku klauzulu. To je trenutak prije odabira klauzule C_2.

$C_2 \equiv$ `uMore(C, M) :- utječeU(C, D), uMore(D, M),`
$\theta_2 = \{$`C` $\leftarrow$ `sava`, `M` $\leftarrow$ `'crno more'`$\}$;
$G_3 \equiv$ `:- utječeU(sava, D), uMore(D, 'crno more');`
$C_3 \equiv$ `utječeU(sava, dunav),`
$\theta_3 = \{$`D` $\leftarrow$ `sava`$\}$;
$G_4 \equiv$ `:- uMore(dunav, 'crno more');`
$C_4 \equiv$ `uMore(C, D) :- utječeU(C, D), more(D),`
$\theta_4 = \{$`C` $\leftarrow$ `dunav`, `D` $\leftarrow$ `'crno more'`$\}$;
$G_5 \equiv$ `:- utječeU(dunav, 'crno more'), more('crno more');`
$C_5 \equiv$ `utječeU(dunav, 'crno more'),`
$\theta_5 = \epsilon$;
$G_6 \equiv$ `:- more('crno more');`
$C_6 \equiv$ `more('crno more');`
$\theta_6 = \epsilon$;
$G_7 \equiv \square$;

Iz teorema adekvatnosti slijedi da je supstitucija $\theta = \theta_0\theta_1\theta_2\theta_3\theta_4\theta_5\theta_6 \restriction_{\texttt{X}}$ supstitucija korektnog odgovora i vrijedi $P \Rightarrow$ `uMore(drava, 'crno more')`. Prolog bi nam sada uzvratio odgovorom „`X = drava`". Sada bismo na isti način mogli dobiti i $P \Rightarrow$ `uMore(sava, 'crno more')`, no primijetimo da bi u tom slučaju Prolog nekoliko puta skrenuo u „ slijepu ulicu", sve dok se ne bi vratio do trenutka prije odabira klauzule C_1.

```
| ?- uMore(X, 'crno more').
X = dunav ? a
X = drava
X = sava
no
```

Još smo prije naglasili da možemo imati programe koji imaju istu semantičku, no različite proceduralne interpretacije. Ako u zadnjoj programskoj klauzuli zamijenimo poredak literala, te nakon toga zamijenimo poredak zadnje dvije programske klauzule, dobit ćemo program koji očito ima isto semantičko značenje (jer su konjunkcija i disjunkcija semantički komutativne), no Prolog neće stati s radom prilikom upita „`uMore(drava, 'crno more')`".

3.2 Složenija rekurzija s pamćenjem prijeđenog puta

Drugi primjer (traženje telefona u kompliciranoj kući s puno soba) pokazuje kako primjenom listi možemo pamtiti proizvoljno komplicirane strukture tijekom traženja. Liste su samo sintaktički istaknute strukture dobivene uzastopnom primjenom binarnog predikata `.`, koji veže početni element i ostatak liste: svaka lista je ili `[]` (prazna), ili oblika `[H|T]`$\equiv$`.(H,T)`, gdje je `H` početni element, a `T` lista preostalih elemenata. Kako je graf ovdje po prirodi neusmjeren (vrata funkcioniraju u oba smjera), za izbjegavanje beskonačnih petlji moramo voditi evidenciju o tome kroz koje sobe smo već prošli, kako ih ne bismo ponavljali u obilasku.

Primjer 26. *[Primjer programa u Prologu]*

```
door(a, b). % +-----—+
door(b, e). % | d c |
door(b, c). % +-- -- —+
door(d, e). % | f e b a
door(c, d). % +-- +-—+
door(e, f). % | g |
door(g, e). % +---—+

hasPhone(g).
go(X, X, _, []).
go(X, Y, T, [Z|Result]):-
(door(X, Z); door(Z, X)),
\+ member(Z,T),
go(Z, Y, [Z|T], Result).
findphone(Path) :-
go(a, Target, [], Path),
 hasPhone(Target).
```

Predikat `go(From, To, Avoid, Result)` izražava činjenicu da je lista `Result` jedna moguća staza od sobe `From` do sobe `To`, izbjegavajući sobe na listi `Avoid`. Znak `;` u klauzuli `(door(X, Z); door(Z, X))` označava logičku disjunkciju: Prolog će pokušati zadovoljiti prvi disjunkt, a nakon backtrackinga (vraćanja na istu točku u DFS obilasku) će pokušati zadovoljiti drugi. To je jednostavan način da zapišemo simetrične relacije: tako možemo u bazi znanja popisati samo veze u jednom smjeru, a u logici programa reći da ih je moguće prelaziti u oba smjera.

Pozivom predikata `findphone` kažemo Prologu da pronađe stazu kojom se moramo kretati da bismo kroz kuću došli do neke sobe s telefonom (u primjeru je to jedino soba `g`). Kao što vidimo, lista "zabranjenih" soba nam omogućuje da tražimo samo jednostavne staze (koje ne ponavljaju sobe), kojih ima konačno mnogo, pa ih Prolog može sve pronaći čak i DFS obilaskom.

```
| ?- findphone(Path).
Path = [b,e,g] ? a
Path = [b,c,d,e,g]
no
```

3.3 Logičko programiranje s uvjetima nad konačnim domenama

Pored direktnog zaključivanja pomoću Hornovih klauzula, Prolog omogućuje i redukciju bilo kakvih uvjeta nad konačnim domenama (s propagacijom ograničenja), što bitno pojednostavljuje programiranje u mnogim slučajevima.

U nekim implementacijama Prologa ta funkcionalnost nalazi se u biblioteci `clpfd` (Constraint Logic Programming over Finite Domains), no GNU Prolog je uključuje u osnovni jezik.

Zadatak. Treba pronaći sve načine na koje $500\,\mathrm{kg}$ brašna možemo rasporediti u vreće od $40\,\mathrm{kg}$ i $60\,\mathrm{kg}$. Primijetimo da je dovoljno zadati uvjete nenegativnosti varijabli: Prolog tada iz jednadžbe zaključuje gornje ograde na njih, iz čega odmah dobiva informaciju da su nad konačnim domenama (implicitna je pretpostavka da su cijeli brojevi). Tada se propagacijom uvjeta lako dobivaju sve mogućnosti.

Primjer 27. *[Primjer programa u Prologu]*

```
vreće(V40, V60) :-
V40 #>= 0, V60 #>= 0,
V40 * 40 + V60 * 60 #= 500,
fd_labeling([V40, V60]).
```

Programiranje je vrlo slično uobičajenom logičkom programiranju u Prologu: popišemo uvjete (svi operatori biblioteke `clpfd` pišu se s početnim znakom `#` da bi se razlikovali od ugrađenih Prologovih operatora), i pomoću `fd_labeling([varijable])` zatražimo rješenja.

```
| ?- vreće(Male, Velike).

Male = 2
Velike = 7 ? a

Male = 5
Velike = 5

Male = 8
Velike = 3

Male = 11
Velike = 1

yes
```

Važno je napomenuti da su parcijalne interpretacije u logici sudova, definirane samo na varijablama koje se pojavljuju u nekom konačnom skupu formula, samo specijalni slučaj varijable nad konačnom domenom (2^n mogućnosti, gdje je n ukupni broj varijabli). To znači da primjenom upravo opisane tehnike možemo rješavati i zadatke sljedećeg tipa.

Zadatak.[2] *Gazdarica na zabavi reče da će besplatni šampanjac dobiti onaj tko ispuni sljedeće uvjete:*

(1)

Ako ne pleše s konobaricom, ili pleše s crnkom, onda ne smije plesati s gazdaricom, ali mora plesati s plavušom.

(2)

Mora plesati s gazdaricom, ali ne smije s konobaricom, osim ako pleše s crnkom, a ne pleše s plavušom.

}

Rješenje zadatka je dano sljedećim Prolog programom. Primijetimo da se pojavljuje logički veznik „osim ako" kojeg Prolog nema definiranog, ali ga možemo lako definirati pomoću implikacije. Deklariranju takvih operatora služi `op` naredba: 740 je razina prioriteta (ista kao za `#==>`), a `xfx` znači da želimo binarni neasocijativni operator.

Primjer 28. *[Primjer programa u Prologu]*

```
:- op(740, xfx, osimako).
A osimako B :- #\ B #==> A.

šampanjac :-
#\ K #\/ C #==> #\ G #/\ P,
G #/\ #\ K osimako C #/\ #\ P,
fd_labeling([G, K, C, P]).
```

Nakon unošenja prethodnog programa, na upit `?- šampanjac.`, Prolog daje odgovor "`no`" (ili "`false.`", ovisno o implementaciji). To znači da gazdaričine uvjete nije moguće ispuniti.

3.4 Malo povijesti za kraj

Na kraju navodimo kratak pregled povijesti logičkog programiranja.

Više detalja možete pročitati u [7].

Iako se ideje o logičkom programiranju mogu pronaći još u intuicionističkoj logici i teoriji dokaza s početka dvadesetog stoljeća, možemo reći da je logičko programiranje kakvog danas poznajemo direktna posljedica ranijeg razvoja automatiziranih dokazivača teorema i umjetne inteligencije iz šezdesetih godina dvadesetog stoljeća. Značajan trenutak u tom razvoju bio je uvođenje metode opće rezolucije (1965) od strane Johna Alana Robinsona. Iako je pravilo rezolucije bilo poznato još prije kod primjene u logici sudova (na primjer kod Davis–Putnamova algoritma), Robinson ga je u svom radu uspio ujediniti s metodom unifikacije, te dokazati adekvatnost i potpunost tako dobivene metode opće rezolucije. Ovaj rezultat je pokrenuo novi val istraživanja automatiziranih dokazivača teorema i mnogi znanstvenici su se okrenuli proučavanju raznih restrikcija pravila opće rezolucije.

Robert Kowalski je 1972. godine došao na ideju da SL–rezoluciju primijeni na Hornovim klauzulama. Takva rezolucija je poslije dobila ime SLD–rezolucija. Na temelju njegovog rada, Alan Colmerauer i Philippe Roussel dizajniraju efikasan logički programski jezik u svrhu procesiranja prirodnog jezika kojeg Roussel naziva Prolog (Programmation Logique). Prvi Prolog interpreter napisao je Roussel u programskom jeziku ALGOL–W, dok se nedugo zatim pojavila poboljšana verzija napisana u programskom jeziku FORTRAN. Prolog je bio dokaz da programski jezik može imati dualnu interpretaciju — deklarativnu od strane programera i proceduralnu od strane parsera.

Velik korak u razvoju Prologa nastao je pojavom Warrenove apstraktne mašine (WAM). Istu je dizajnirao David Warren u svrhu efikasnijeg izvršavanja Prolog programa. WAM je danas postao meta skoro svih Prolog prevoditelja. Na razvoj Prologa također je utjecao i FGCS (eng. Fifth Generation Computer Systems Project). Paralelizirana varijanta Prologa pod imenom KL1 (eng. Kernel Language 1) korištena je kao jezik operativnog sustava za FGCS.

Prolog je danas daleko najpopularniji logički (odnosno deklarativni) programski jezik. Postoje mnoge kvalitetne Prolog implementacije. Neke od njih su GNU Prolog, SWI–Prolog, Visual Prolog i YAP–Prolog. Prolog je, također, utjecao na pojavu novih programskih jezika. Neki od značajnijih su DATALOG, Mercury, Gödel i Erlang.

Bibliografija

[1]

M. Ben–Ari, *Mathematical Logic for Computer Science*, 3rd Edition, Springer, 2012.

[2]

A. Haken, *The Intractability of Resolution*, Theor. Comput. Sci., 39 (1985), 297–308

[3]

J. W. Lloyd, *Foundations of Logic Programming*, 2nd Edition (Extended), Springer–Verlag, 1987.

[4]

A. Martelli, U. Montanari, *An Efficient Unification Algorithm*, ACM Trans. Program. Lang. Syst., 4(1982), 258–282

[5]

M. Mihelčić, *Davis–Putnamov algoritam*, diplomski rad, PMF–MO, Zagreb, 2011.

web.math.pmf.unizg.hr/∼vukovic/Diplomski-radovi/Mihelcic-Davis-Putnamov- algoritam.pdf

[6]

M. Mihelčić, T. Lolić, *Problem ispunjivosti logičke formule (SAT)*, math.e, 21, 2012.

[7]

P. Paradžik, *Logičko programiranje*, diplomski rad, PMF–MO, Zagreb, 2014.

[8]

L. Sterling, E. Shapiro, *The Art of Prolog: Advanced Programming Techniques*, 2nd Edition, MIT Press, 1994.

[9]

M. Vuković, *Matematička logika*, Element, Zagreb, 2009.

[10]

M. Vuković, *Izračunljivost*, skripta, PMF–MO, Zagreb, 2009.

[1]Smatramo da je čitatelju poznat pojam terma. Formalnu definiciju terma, kao i ostalih ovdje nedefiniranih pojmova u vezi logike prvog reda, možete vidjeti, primjerice, u [9].

[2]Ovo je verzija jednog zadatka D. Blanuše. Više o tome možete pronaći u [9].

ISSN 1334-6083