Gilmorova procedura za rezonovanje u logici prvog reda zasnovana na Erbranovoj teoremi

Autor: Marinela Parović

Profesor: Filip Marić

Sadržaj

1 Erbranova teorema

U logici prvog reda nije odlučivo da li je neka formula valjana, što znači da ne postoji opšti postupak (procedura ili algoritam) kojim se za proizvoljnu formulu logike prvog reda može ispitati da li je valjana i dobiti potvrdan ili odričan odgovor.

Međutim, moguće je formulisati postupak koji će biti u mogućnosti da za svaku valjanu formulu pokaže da je valjana. Zato kažemo da je pitanje valjanosti u logici prvog reda poluodlučiv problem. Erbranova teorema daje mogućnost ispitivanja poluodlučivosti, uspostavljanjem veze između logike prvog reda i iskazne logike. Formule logike prvog reda bez kvantifikatora se mogu posmatrati kao formule iskazne logike, uz proširenje da se umjesto iskaznih slova pojavljuju atomi logike prvog reda. Može se dokazati da važi sljedeći stav.

Stav 1 *Formula logike prvog reda je valjana akko je iskazna tautologija.*

Željeli bismo da pronađemo sličnu vezu i za zadovoljivost u logici prvog reda i u iskaznoj logici. Ta veza je bitna jer se postupkom skolemizacije dobija ekvizadovoljiva, a ne ekvivaljana formula. Za formule koje su rečenice (bazne formule ili formule bez promjenljivih) važi sljedeći stav.

Stav 2 *Bazna formula bez kvantifikatora logike prvog reda je zadovoljiva akko je iskazno zadovoljiva.*

Za proizvoljnu formulu bez kvantifikatora važi sljedeće.

Stav 3 *Ako je formula bez kvantifikatora logike prvog reda zadovoljiva, onda je ona i iskazno zadovoljiva.*

Međutim, obrnut smjer ne važi.

Definicija 1 *Skup svih baznih termova jezika $\mathcal{L}$ nazivamo Erbranov univerzum i označavamo sa $H(\mathcal{L})$.*

Ukoliko jezik $\mathcal{L}$ ne sadrži nijednu konstantu u njega se umeće novi simbol konstante kako Erbranov univerzum ne bi bio prazan.

Erbranov univerzum formule F u oznaci $H(F)$ je Erbranov univerzum jezika sačinjenog od simbola koji se javljaju u toj formuli.

Definicija 2 *Bazne instance formule su formule koje se dobijaju supstitucijom promjenljivih elementima Erbranovog univerzuma te formule.*

Sljedeća teorema formuliše kriterijum koji daje vezu između zadovoljivosti u logici prvog reda i u iskaznoj logici.

Teorema 1 *(Erbran) Formula bez kvantifikatora F je zadovoljiva akko je skup svih njenih baznih instanci iskazno zadovoljiv. Preciznije, formula F oblika $\forall x_1 ... x_n . \Phi(x_1, ..., x_n)$ je zadovoljiva akko je zadovoljiv skup*

$$\{\Phi[x_1 \to t_1]...[x_n \to t_n] | t_1, ..., t_n \in H(F)\}.$$

Teorema 2 *(Erbran) Erbranova interpretacija zadovoljava formulu bez kvantifikatora akko zadovoljava sve njene bazne instance.*

2 Gilmorova procedura

Na osnovu tvrđenja Erbranove teoreme ispitivanje zadovoljivosti formule bez kvantifikatora se svodi na ispitivanje zadovoljivosti skupa svih baznih instanci. Na osnovu teoreme kompaktnosti imamo da je formula bez kvantifikatora nezadovoljiva akko je postoji konačan nezadovoljiv podskup skupa svih baznih instanci.

Procedure Erbranovog tipa nabrajaju skupove baznih instanci dodajući nove instance. Takva je i Gilmorova procedura. U njoj se nezadovoljivost tekućeg skupa baznih instanci ispituje prevođenjem u DNF. Dakle, na osnovu elemenata Erbranovog univerzuma u skup baznih instanci se dodaju nove instance, koje se sa prethodno generisanim instancama povezuju konjunkcijom. Zatim se taj tekući skup baznih instanci prevodi u DNF. Formula koja je u DNF je nezadovoljiva ako je svaka od klauza netačna. Da bi se to provjerilo, potrebno je u svakoj klauzi pronaći bar jedan par suprotnih literala.

Ukoliko polazna formula nije valjana, nove bazne instance će se "beskonačno" dodavati u tekući skup baznih instanci, i procedura će se "beskonačno" izvršavati. Inače, ukoliko je polazna formula valjana, njena negacija je nazadovoljiva. Na osnovu teoreme kompaktnosti, znamo da postoji konačan nezadovoljiv skup baznih instanci, pa se Gilmorova procedura završava nakon konačnog broja koraka. Ovim je obrazložena poluodlučivost ove procedure, koja je ranije i najavljena.

Gilmorova procedura je jako neefikasna. Prevođenje formule u DNF može eksponencijalno uvećati formulu. Zbog toga, ovom procedurom se ne može dokazati valjanost mnogih formula koje jesu valjane u razumnom vremenu.

3 Implementacija

Gilmorova procedura implementirana je u programskom jeziku C++. Implementaciji same procedure prethodila je implementacija sintakse i semantike logike prvog reda. Implementirani su signatura, domen i $\mathcal{L}$-struktura (deklaracije su u `signature.h`, `domain.h` i `lstructure.h`). Dalje je razvijena hijerarhija klasa kojima se predstavljaju termovi u vidu bazne apstraktne klase `BaseTerm` i njenih potklasa `VariableTerm` i `FunctionTerm` kojima se predstavljaju promjenljive i funkcijski termovi (deklaracije su u `baseterm.h`, `variableterm.h` i `functionterm.h`. Dalje je uslijedila implementacija valuacije koja dodjeljuje promjenljivima vrijednosti tačno ili netačno (`valuation.h`). Bazna apstraktna klasa hijerarhije kojom su predstavljene formule je `BaseFormula` (`baseformula.h`). Atomične formule implementirane su kroz apstraktnu baznu klasu `AtomicFormula` (`atomicformula.h` i njene konkretne instance `True`, `False` (deklaracije su u `constants.h`) i `Atom` (`atom.h`). Negacija je predstavljena klasom `Not` (`not.h`) koja nasljeđuje apstraktnu klasu unarnih veznika `UnaryConnective` (`unaryconnective.h`). Binarni veznici konjunkcija, disjunkcija, implikacija i ekvivalencija realizovani su redom klasama `And`, `Or`, `Imp` i `Iff` (deklaracije su u datotekama `and.h`, `or.h`, `imp.h` i `iff.h`). Ove četiri klase nasljeđuju klasu `BinaryConnective` kojoj je predstavljen binarni veznik (`binaryconnective.h`). Egzistencijalni i univerzalni kvantifikator predstvaljeni su klasama `Exists` i `Forall` (`exists.h` i `forall.h`) i oni nasljeđuju baznu klasu Quantifier kojom se predstavlja kvantifikator (`quantifier.h`). Još neke deklaracije i uvođenja novih imena za postojeće tipove nalaze se u datoteci `common.h`.

Klasa `HerbrandUniverse` predstavlja Erbranov univerzum za datu formulu i signaturu (`herbrand_universe.h`).

```
#ifndef HERBRAND_UNIVERSE_H
#define HERBRAND_UNIVERSE_H
#include "first_order_logic.h"
#include "utils.h"

/*
    Klasa za generisanje Erbranovog univerzuma za datu
    signaturu i formulu.
*/

class HerbrandUniverse {
public:
    /* Konstruktor koji pravi objekat za datu
    signaturu i formulu. */
    HerbrandUniverse(const Signature::Sptr& signature,
    const Formula& formula);

    /* Metod koji generise sljedeci nivo E(i+1)
    Erbranovog univerzuma tako da i svi prethodni
    elementi univerzuma budu u njemu, tj. E(i) je
    podskup od E(i+1). */
    void nextApplication();

    /* Get metod za Erbranov univerzum. */
```

```
25          const std::set<Term>& universe() const;

27          /* Metod koji stampa elemente univerzuma na zadati
            ostream. */
29          std::ostream& print(std::ostream& out) const;

31      private:
            /* Signatura */
33          Signature::Sptr m_signature;

35          /* Formula */
            Formula m_formula;
37
            /* Skup funkcijskih simbola formule ukljucujuci i
39          konstante. */
            FunctionSet m_functions;
41
            /* Erbranov univerzum predstvljamo kao skup termova. */
43          std::set<Term> m_universe;

45          /* Ova clanica ce sadrzati generisanu konstantu,
            ako se u formuli ne javlja nijedna konstanta. */
47          FunctionSymbol m_constant = "";
};
49
/* Operator << za ispis elemenata Erbranovog univerzuma. */
51 std::ostream& operator<<(std::ostream& out, const HerbrandUniverse&
       universe);

53 #endif // HERBRAND_UNIVERSE_H
```

Listing 1: Implementacija Gilmorove procedure

Datoteka `herbrand_universe.cpp` sadrži definicije metoda za Erbranov univerzum. Slijedi prikaz konstruktora koji vrši inicijalizaciju Erbranovog univerzuma, kao i metoda `nextApplication()` koji generiše sljedeći nivo Erbranovog univerzuma. Prilikom generisanja sljedećeg nivoa koristi se funkcija `variations_with_repetition` koja generiše varijacije sa ponavanjem elemenata datog skupa i zadate dužine.

```
    HerbrandUniverse::HerbrandUniverse(const Signature::Sptr &
    signature, const Formula &formula)
2   : m_signature(signature), m_formula(formula)
{
4   /* Nalaze se funkcijski simboli u formuli, ukljucujuci i
    konstante. */
    m_formula->getFunctions(m_functions);
6
    /* Inicijalno se u Erbanov univerzum dodaju samo konstante iz
    formule. */
8   for(const FunctionSymbol& fsymb : m_functions)
        if(signature->getFunctionArity(fsymb) == 0)
10          m_universe.insert(std::make_shared<FunctionTerm>(
    m_signature, fsymb));

12  /* Ako formula ne sadrzi nijednu konstantu, dodaje se nova
    konstanta u Erbranov univerzum i u signaturu. */
```

```
    if(m_universe.empty()) {
        m_constant = m_signature->getUniqueFunctionSymbol();
        m_signature->addFunctionSymbol(m_constant, 0);
        m_universe.insert(std::make_shared<FunctionTerm>(
    m_signature, m_constant));
    }
}
void HerbrandUniverse::nextApplication()
{
    /* Vektor za cuvanje svih varijacija sa ponavljanjem. */
    std::vector<std::vector<Term>> variations;
    /* Kopija prethnodnog univerzuma. */
    std::set<Term> universeCopy;
    universeCopy.insert(m_universe.cbegin(), m_universe.cend());
    m_universe.clear();

    /* Ako je u univerzum bila dodata nova konstanta dodajemo je
    ponovo. */
    if(m_constant != "")
        m_universe.insert(std::make_shared<FunctionTerm>(
    m_signature, m_constant));

    /*Za svaki funkcijski simbol koji se pojavljuje u formuli... */
    for(const FunctionSymbol& fsym : m_functions) {
        variations.clear();
        unsigned length = m_signature->getFunctionArity(fsym);
        /* ... arnosti length, generisu se sve varijacije
        sa ponavljanjem elemenata prethodnog Erbranovog
        univerzuma duzine length... */
        std::vector<Term> current(length);
        variations_with_repetition<Term>(0, length,
        current, universeCopy, variations);

        /* ... i u Erbranov univerzum se dodaje term koji
        nastaje primjenom tog funkcijskog simbola na svaku
        od dobijenih varijacija. */
        for(const auto& v : variations)
            m_universe.insert(std::make_shared<FunctionTerm>(
    m_signature, fsym, v));
    }
}
```

Listing 2: Implementacija Gilmorove procedure

Implementacija Gilmorove procedure nalazi se u datotekama `gilmore_procedure.h` i `gilmore_procedure.cpp` i predstavljena je funkcijom čija je deklaracija, a zatim i implementacija navedena u nastavku.

```
#include "first_order_logic.h"
#include "herbrand_universe.h"

#define MAX_ITERATIONS 3

/* Nabrojivi tip koji je povratna vrijednost Gilmorove procedure.
    */
enum ProcedureState {
    VALID,
```

```
    MAX_ITERATIONS_REACHED
};
/* Glavna funkcija koja implementira Gilmorovu proceduru. Funkcija
    prihvata signaturu i formulu f te signature za koju se dokazuje
     da je valjana. Povratna vrijednost je: VALID - ako je f
    valjana ili MAX_ITERATIONS_REACHED - ako je dostignut
    maksimalni dozvoljeni broj iteracija prije nego sto je dokazano
     da je formula valjana. U funkciji se f negira, zatim se vrsi
    skolemizacija, a onda i uklanjanje univerzalnih kvantifikatora
    sa pocetka formule. Nakon toga se, korak po korak, generise
    Erbranov univerzum formule i na osnovu njega se prave bazne
    instance, koje se spajaju konjunkcijom. Tako dobijena formula
    se prevodi u DNF, i provjerava se da li su sve konjunkcije u
    DNF-u netacne. Ukoliko jesu, vraca se VALID, a inace se
    generise sljedeci nivo Erbranovog univerzuma. Opisani korak se
    ponavlja najvise MAX_ITERATIONS puta.
 */
ProcedureState gilmoreProcedure(const Signature::Sptr& signature,
    const Formula& f);
```

Listing 3: Implementacija Gilmorove procedure

```
#include "gilmore_procedure.h"
#include <algorithm>

ProcedureState gilmoreProcedure(const Signature::Sptr &signature,
    const Formula &f)
{
    /* Formula f cija se valjanost dokazuje se negira. */
    Formula notF = std::make_shared<Not>(f);

    /* Negirana formula se prevodi u Skolemovu normalnu formu. */
    Formula skolemNotF = notF->simplify()->nnf()->prenex()->skolem(
    signature);

    /* Sa pocetka formule f se uklanjaju eventualni univerzalni
    kvantifikatori. */
    Formula fBase = removeUniversalFromSkolem(skolemNotF);

    /* Formira se Erbranov univezum formule f. */
    HerbrandUniverse universe(signature, f);

    /* Nalazi se skup promjenljvih koje se pojavljuju u formuli
    nakon skolemizacije, i taj skup se kopira u vektor zbog
    indeksiranja prilikom supstitucije.
    */
    VariablesSet vset;
    f->getVars(vset);
    std::vector<Variable> variables;
    std::copy(vset.cbegin(), vset.cend(), std::back_inserter(
    variables));

    /* Promjenljiva nVars je broj promjenljivih u formuli nakon
    skolemizacije. */
    unsigned nVars = variables.size();

    /* MAX_ITERATIONS puta se ponavlja sljedeci korak... */
```

```
    for(unsigned iter = 0; iter < MAX_ITERATIONS; iter++) {
30      /* Racuna se skup varijacija sa ponavljanjem
        duzine nVars od elemenata Erbranovog univerzuma i
32      tako dobijeni skup se cuva u vektoru variations. */
        std::vector < std::vector<Term> > variations;
34      std::vector <Term> current(nVars);
        variations.clear();
36      variations_with_repetition<Term>(0, nVars, current,
    universe.universe(), variations);

38      /* Formira se formula F koja ce biti konjukcija
        svih baznih instanci dobijenih supstitucijom
40      promjenljivih iz formule fBase svakom od
        izracunatih varijacija sa ponavljanjem. */
42
        /* Na pocetku, F prestavlja formulu nastalu
44      supstitucijom promjenljivih termovima prve
        varijacije... */
46      Formula F = fBase;
        for(unsigned i = 0; i < nVars; i++)
48          F = F->substitute(variables[i], variations[0][i]);

50      /* ...a zatim se na F konjukcijom nadovezuju i
        formule Fi nastale supstitujom svim ostalim
52      varijacijama. */
         for(unsigned i = 1; i < variations.size(); i++) {
54          Formula Fi = fBase;
            for(unsigned j = 0; j < nVars; j++)
56              Fi = Fi->substitute(variables[j],
                variations[i][j]);
58          F = std::make_shared<And>(F, Fi);
        }
60
        /* Nalazi se DNF tako dobijene formule. */
62      LiteralListList dnfF = F->listDNF();

64      /* Brojac koji ce biti jednak broju netacnih
        konjunkcija u DNF-u. */
66      unsigned nFalseConjuctions = 0;

68      /* Za svaku konjukciju u DNF-u... */
        for(const LiteralList& conjunction : dnfF) {
70          /* Vektor nenegiranih atoma u trenutnoj konjunkciji. */
            LiteralList positive;
72
            /* Vektor negiranih atoma u trenutnoj konjunkciji. */
74          LiteralList negative;
             /* Za svaki literal u konjunkciji... */
76          for(const Formula& literal : conjunction) {
                /* Ako je literal negacija atoma... */
78              if(BaseFormula::isOfType<Not>(literal)) {
                    const Not* notF = static_cast<const Not*>(
    literal.get());
80                  /*... ako postoji atom koji je jednak
                    njemu u vektoru nenegiranih atoma
82                  trenutna konjunkcija je netacna, pa se
                    brojac uvecava i prelazi se na
```

```
84                      sljedecu zbog break naredbe. */
                        if(std::find_if(positive.begin(), positive.end
    (), [&](const auto& fla) {
86                          return notF->operand()->equalTo(fla);
                                }) != positive.end()) {
88                          nFalseConjuctions++;
                            break;
90                      }
                        /* inace se operand negiranog atoma
92                      dodaje u vektor negiranih atoma
                        formule. */
94                      else
                            negative.push_back(notF->operand());
96                  }
                    /* Ako je literal atom... */
98                  else {
                        /*... ako postoji atom koji je jednak
100                     njemu u vektoru negiranih atoma
                        trenutna konjunkcija je netacna, pa se
102                     brojac uvecava i prelazi se na
                        sljedecu zbog break naredbe. */
104                      if(std::find_if(negative.begin(), negative.end
    (),
                                        [&](const auto& fla) {
106                                         return literal->equalTo(fla
    );
                                        }) != negative.end()) {
108                         nFalseConjuctions++;
                            break;
110                     }
                        /* inace se atom dodaje u vektor
112                     nenegiranih atoma formule. */
                        else
114                         positive.push_back(literal);
                    }
116             }
                /* Vektori se prazne da bi bili spremni za
118             sljedecu konjunkciju. */
                positive.clear();
120             negative.clear();
            }
122
            /* Ako je broj netacnih konjunkcija jednak ukupnom
124         broju konjunkcija, formula je nezadovoljiva, pa je
            f valjana. */
126          if(nFalseConjuctions == dnfF.size())
                return VALID;
128         /* inace se prelazi na sljedeci nivo Erbranovog
            univerzuma. */
130         universe.nextApplication();
        }
132     return MAX_ITERATIONS_REACHED;
    }
```

Listing 4: Implementacija Gilmorove procedure