Automatsko rezonovanje - vežbe

Sadržaj

Čas 1 - iskazna logika, formiranje drveta izraza

Preduslovi: *osnovno poznavanje programskog jezika C++; nasleđivanje u programskom jeziku C++; poznavanje standardne C++ biblioteke;*

Osnove iskazne logike

Kada govorimo prirodnim jezikom izražavamo se rečenicama koje su sastavljene od reči, dakle na neki način rečenice su *izrazi* koje koristimo. Slično, ako je u pitanju niz simbola “5 + 6 - 1 * 3” on se prirodno prepoznaje kao aritmetički izraz. Kada nešto računamo radimo sa aritmetičkim izrazima.
Što se tiče iskazne logike početni cilj je sličan - potrebno je definisati i formirati izraze iskazne logike sa kojima ćemo u nastavku raditi.

Def 1.1: Izrazi iskazne logike su iskazne formule koje se na nivou apstraktne sintakse grade:

1. *od iskaza - T (tačno) i F (netačno), iskaznih slova (nazivaju se još i atomi, odnosno iskazne promenljive)*
2. *primenom logičkih veznika ~, /\, \/, => i <=> na iskaze*

Formalna definicija skupa iskaznih fomula govori o tome kako je to najmanji skup reči iznad najviše prebrojivog skupa atoma P, međutim mi ćemo se fokusirati na definiciju koja je malo više prilagođena našim potrebama, a to je implementacija programa za rad sa izrazima iskazne logike.

Def 1.2: Skup iskaznih formula se može opisati pomoću sledeće gramatike:

Formula -> *T*
| *F*
| *p ∈ P (gde je p iskazno slovo odnosno atom)*
| *~(Formula)*
| *(Formula) /\ (Formula)*
| *(Formula) \/ (Formula)*
| *(Formula) => (Formula)*
| *(Formula) <=> (Formula)*

Iz prethodne *opisne* definicije sledi da je rekurzivno rešenje jedno prirodno rešenje koje ćemo i mi iskoristiti - predstavljaćemo formule iskazne logike kao drvo izraza.

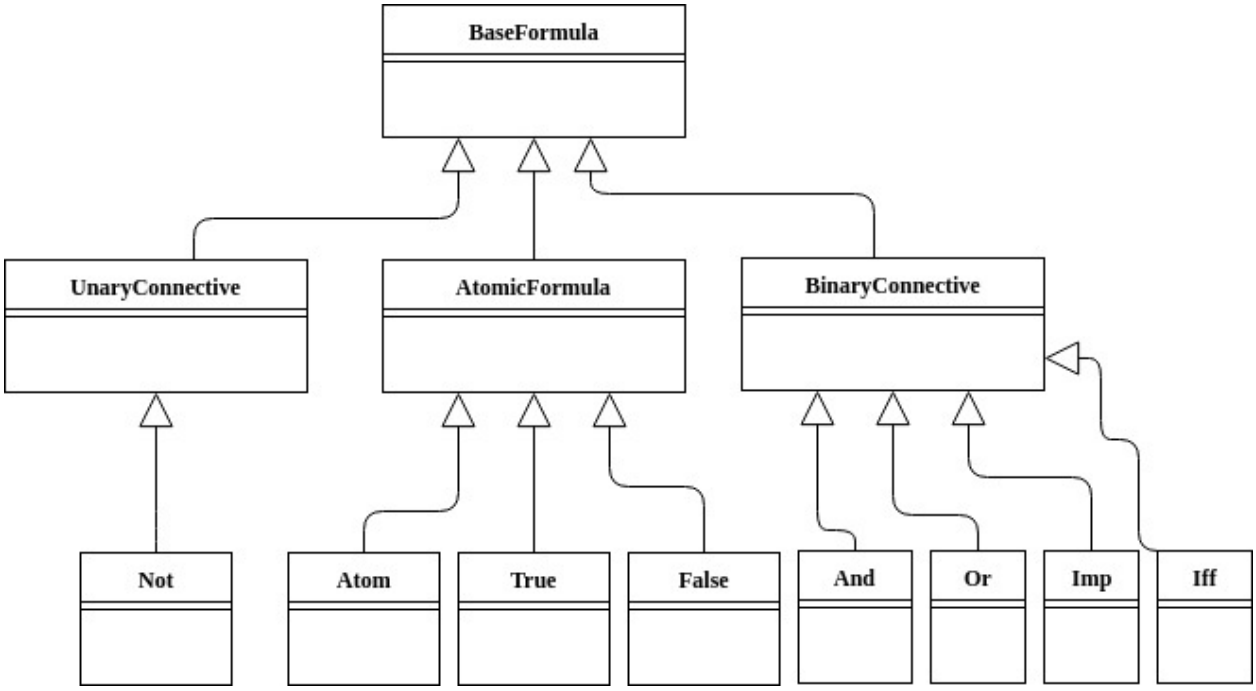
Hijerarhija klasa za predstavljanje logičkih izraza

Osnovni cilj je dobiti strukturu takvu da se sa raznorodnim iskaznim formulama, kao što su na primer *~p* i *(p /\ q \/ ~r) => ~q*, može raditi na isti način. Dakle, svakako nam treba bazna klasa da operišemo sa formulama različitog tipa kroz pokazivač (ili referencu) na baznu klasu.
Dalje, ako se navedena opisna definicija razmatra malo pažljivije, mogu se uočiti bar još tri apstraktna koncepta:

1. jednostavne formule bez logičkih veznika (tj. preciznije *atomičke formule*), koje uključuju T, F i iskazna slova
2. formule sa unarnim logičkim veznikom, trenutno je to samo ~(Formula) (međutim treba uzeti u obzir činjenicu da je mogla da postoji još neka unarna operacija da se radi o nekom drugom domenu koji nije iskazna logika)

3. formule sa binarnim logičkim veznikom, imamo po jedan takav slučaj za veznike /\, \/, => i <=>

Da bilo jasnije zašto tražimo generalizaciju koja je zajednička za više konkretnih klasa (npr. binarni logički veznik je generalizacija koja se odnosi /\, \/, => i <=>) razmotirmo implementaciju jednostavne operacije koja za prosleđenu formulu vraća složenost formule. Složenost formule se meri brojem logičkih veznika koji učestvuju u izgradnji formule. Na primer, formula ~p ima složenost 1, formula T složenost 0, a formula ~(p /\ q) složenost 2. Umesto da za svaki od binarnih veznika implementiramo ovaj metod posebno postoji rekurzivna implementacija koja uopšte ne uzima u obzir koji je binarni veznik u pitanju: *return 1 + complexity(operand1) + complexity(operand2)* (gde su *operand1* i *operand2* podformule direktno spojene datim binarnim veznikom). Ovakve generalizacije nam omogućavaju da pišemo manje koda koji je ujedno i lakši za održavanje.
Hijerarhija klasa koju ćemo koristiti za predstavljanje logičkih izraza data je u Dijagramu 1.1.



Dijagram 1.1: Hijerarhija klasa za predstavljanje iskaznih formula

Valuacija; interfejs bazne klasne; deljeni pokazivači u C++-u

Pre nego što pogledamo interfejs bazne klase pozabavimo se još jednim konceptom - funkcijom valuacije. Dakle, iskazne formule imaju svoju vrednost koja je istinitosna tj. tačno ili netačno i zavisi samo od vrednosti iskaznih promenljivih (atoma).

Def 1.3: Valuacija je funkcija koja sve atome preslikava u dvočlani skup {0, 1} pri čemu se podrazumeva da ako je vrednost atoma 1, onda je on tačan, a inače je netačan.

Dakle formula ima vrednost u zavisnosti od trenutne valuacije (trenutne vrednosti atoma). Iz gornje definicije se takođe vidi da se valuacija može jednostavno predstaviti pomoću mape. Formule *(p /\ q) \/ r* i $(p_0 \wedge p_1) \vee p_2$ su potpuno iste do na upotrebu simbola. Jedno prirodno rešenje za predstavljanje atoma je da se oni predstave *unsigned* tipom gde na primer broj 3 odgovara atomu p_3. U takvom kodiranju valuacija se jednostavno predstavlja sa *std::map<unsigned, bool>* gde se svakom atomu dodeljuje vrednost *true* ili *false*.
Sledi osnovni interfejs bazne klase koji ćemo proširivati po potrebi kasnije:

```
class BaseFormula;
using Formula = std::shared_ptr<BaseFormula>;

class BaseFormula : public std::enable_shared_from_this<BaseFormula> {
public:
 BaseFormula();
  virtual std::ostream& print(std::ostream &out) const = 0;
  virtual unsigned complexity() const = 0;
  virtual Formula substitute(const Formula &what, const Formula &with) const = 0;
  virtual bool eval(const Valuation &val) const = 0;
  virtual void getAtoms(AtomSet &aset) const;
  virtual bool equalTo(const Formula &f) const;
  virtual ~BaseFormula();
};
bool operator==(const Formula &lhs, const Formula &rhs);
bool operator!=(const Formula &lhs, const Formula &rhs);
std::ostream& operator<<(std::ostream &out, const Formula &f);
```

Što se samih metoda bazne klase tiče njihova imena dobro određuju njihovu funkcionalnost (npr. *eval()* vraća vrednost formule pri prosleđenoj valuaciji, *getAtoms()* dodaje atome koji se nalaze u tekućoj formuli u prosleđeni skup atoma itd.).

Ne baš očigledan deo se odnosi na definiciju samog tipa formule:

```
using Formula = std::shared_ptr<BaseFormula>;
```

pa ćemo mu posvetiti malo pažnje. Fokusirajmo se na primer na binarni veznik (klasa *BinaryConnective* u nastavku). Jasno je da ova klasa mora sadržati dve formule kao članice klase. Takođe bi trebalo da bude jasno da ove članice moraju biti pokazivačkog tipa, na primer:

```
BaseFormula *m_op1, *m_op2;
```

Dalje, posmatrajmo jednostavnu formulu: $(\underline{p_0} \wedge p_1) \vee \underline{p_0}$. Formula koja je tipa *Atom* se dva puta javlja za atom p_0. Dakle, postoji formula tipa *Or* koja sadrži referencu na formulu koja se odnosi na p_0 i postoji formula tipa *And* koja sadrži referencu na p_0, formula data u pseudokodu je:

```
a0 = Atom(0);
f = Or(And(a0, Atom(1)), a0)
```

S obzirom na to da 2 formule sadrže isti pokazivač, postavlja se pitanje koja od te dve formule treba da oslobodi memoriju za formulu tipa *Atom* koja se odnosi na *p0*? Takođe, šta se dešava ako *N* različitih formula referiše na neku podformulu *f*, ko je tada odgovoran za oslobađanje memorije?

Problem upravljanja memorijom se u programskom jeziku C++ najelegantnije rešava upotrebom *pametnih pokazivača* (eng. *smart pointers*). U pitanju su šablonske klase *std::shared_ptr<T>* (deljeni pokazivač u nastavku) i *std::unique_ptr<T>* (jedinstveni pokazivač u nastavku). Prva se koristi u slučaju koji je opisan iznad, kada više instanci referiše na jedan pokazivač. Deljeni pokazivač funkcioniše tako što čuva pokazivač i još jedan broj koji predstavlja ukupan broj trenutnih objekata koji referišu na taj pokazivač. Svaki put kada se deljeni pokazivač kopira (konstruktorom kopije ili operatorom dodele), ovaj broj se uveća za jedan. Slično, svaki put kada se pozove destruktor za neku od kopija deljenog pokazivača ovaj broj se smanji za jedan. Kada je vrednost brojača referenci 0 tada se memorija na koju pokazivač pokazuje oslobađa. Primer koncepta implementacije deljenog pokazivača (prava implementacija je daleko komplikovanija):

```
class SharedPointerToMyClass {
public:
 SharedPointerToMyClass() {
   m_shared = new MyClass{};
```

```
    m_counter = new unsigned;
    *m_counter = 1U;
  }
   SharedPointerToMyClass(const SharedPointerToMyClass &oth) {
    m_shared = oth.m_shared;
    m_counter = oth.m_counter;
    *m_counter += 1;
  }
   SharedPointerToMyClass& operator=(const SharedPointerToMyClass &oth) {
    if (this != &oth) {
      /* Smanjujemo brojac i oslobadjamo memoriju ako je brojac 0 */
      *m_counter -= 1;
      if (0 == *m_counter) {
        delete m_shared;
        delete m_counter;
      }
      /* Kopiramo podatke i uvecavamo brojac referenci */
      m_counter = oth.m_counter;
      m_shared = oth.m_shared;
      *m_counter += 1;
    }
    return *this;
  }
   ~SharedPointerToMyClass() {
    *m_counter -= 1;
    if (0U == *m_counter) {
      delete m_shared;
      delete m_counter;
    }
  }
private:
  MyClass *m_shared;
  unsigned *m_counter;
};
```

Jedan detalj je još uvek nerazjašnjen - upotreba *std::enable_shared_from_this<T>*. Pokazaće se da korektna implementacija zahteva da se prilikom implementacije određenih funkcija članica izvrši naredba *return std::shared_ptr<T>(this)* koja je problematična iz razloga koji su van obima i cilja ovog kursa. Nasleđivanje klase *std::enable_shared_from_this<BaseFormula>* rešava ovaj problem. Zainteresovani čitalac može detalje naći na ovoj Veb strani.

Što se tiče jedinstvenog pokazivača, za njega ne postoji konstruktor kopije niti postoji operator dodele - cela poenta je u tome da se sa sigurnošću obezbedi da postoji samo jedna referenca na memoriju na koju pokazuje pokazivač. Dodatno, upotreba *std::shared_ptr<T>* i *std::unique_ptr<T>* nam garantuje da će resursi biti ispravno oslobođeni u slučaju izuzetaka zbog toga što će se pozvati destruktor. Sada bi bio zgodan trenutak za citaoce da se vrate na interfejs bazne klase i pogledaju ga još jednom - ovaj put detaljnije.

Zadovoljivost, logička ekvivalencija i logička posledica

Već je rečeno da iskazne formule u zavisnosti od valuacije imaju istinitosnu vrednost tačno ili netačno. U tom duhu proširujemo interfejs bazne klase na sledeći način:

```
using OptionalValuation = std::experimental::optional<Valuation>;
class BaseFormula : public std::enable_shared_from_this<BaseFormula>
{
public:
        …
  OptionalValuation isSatisfiable() const;
  OptionalValuation isNotTautology() const;
  bool isConsequence(const Formula &f) const;
  bool isEquivalent(const Formula &f) const;
  void printTruthTable(std::ostream &out) const;
};
```

Klasa *std::experimental::optional<T>* omogućava korisniku klase da kao povratnu vrednost funkcije vrati ili vrednost objekta tipa *T* ili informaciju o tome da takvog objekta nema. Na primer, ista funkcionalnost bi mogla da se pruži i sa sledećim potpisima metoda:

```
Valuation isSatisfiable(bool &isSatisfiable) const;
bool isSatisfiable(Valuation &isSatisfiable) const;
```

Moglo bi se reći da je upotreba *std::experimental::optional<T>* malo elegantnija od predstavljenih alternativa, primer upotrebe bi izgledao ovako:

```
OptionalValuation vopt = f->isSatisfiable();
if(vopt)
{
  cout << "SAT: " << vopt.value() << endl;
}
```

Što se dodatih metoda tiče *isSatisfiable()* proverava da li je formula zadovoljiva i vraća valuaciju koja je zadovoljava ukoliko ona to jeste, *isNotTautology()* vraća valuaciju za koju formula nije tačna (ako takve nema formula je tautologija), *isConsequence()* proverava da li je prosleđena formula *f* posledica tekuće formule, *isEquivalent()* proverava da li su formule logički ekvivalentne i na kraju *printTruthTable()* štampa istinitosnu tablicu za datu formulu. Ove osobine formula su formalno date sledećim definicijama:

Def 1.4: Formula f je zadovoljiva ukoliko postoji valuacija v u kojoj je formula tačna. Dodatno kažemo da je v model za f.

Def 1.5: Formula f je tautologija ukoliko je tačna pri svakoj valuaciji.

Def 1.6: Formule f1 i f2 su logički ekvivalentne ukoliko je svaki model za f1 ujedno model za f2 i obrnuto.

Def 1.7: Formula f2 je logička posledica formule f1 ukoliko je svaki model za f1 ujedno model za f2.

Istinitosna tablica za neku formulu predstavlja njenu vrednost za sve kombinacije vrednosti atoma. Na primer za formulu *(p /\ q) \/ r*:

p	q	r	(p /\ q) \/ r
0	0	0	0

0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Implementacija

Detalji na koje treba obratiti pažnju

Generisanje naredne valuacije

Algoritam za generisanje naredne valuacije je prilično jednostavan. Pogledajmo par primera:
0 0 1 0 1 -> 0 0 1 1 0
0 1 1 1 0 -> 0 1 1 1 1
0 0 1 1 1 -> 0 1 0 0 0
Podvučene nule su ključ elegantnog algoritma. Krećemo se s desna na levo i invertujemo sve vrednosti dok ne dođemo do prve nule, invertujemo još i nju i tu stajemo. Kada je permutacija sastavljena od svih jedinica, nemamo narednu permutaciju, preciznije naredna permutacija je opet početna.

Provera da li su dve formule istog tipa

Da bi dve formule bile sintaksno jednake one pre svega moraju biti istog tipa. U pokazivaču bazne klase *BaseFormula** kao i u *Formula* (*std::shared_ptr<BaseFormula>*) se mogu čuvati instance bilo koje izvedene klase. Kada imamo *Formula f1* i *Formula f2* jedna je možda instanca *And* klase, dok je druga na primer instanca *Or* klase. Programski jezik C++ ima dva ugrađena mehanizma za dinamičku proveru da li su dva objekta istog tipa:

1. *typeid()* operator - vraca informacije o tipu odnosno *std::type_info* koji se može porediti na jednakost (za upotrebu pogledati implementaciju metoda *equalTo()* u klasi *BaseFormula*).
2. *dynamic_cast<>()* - služi za bezbedno kastovanje natklase u potklasu; ukoliko referenca natklase ne referiše na traženu potklasu *dynamic_cast* će vratiti *nullptr*.

Često se koristi još jedan način da se obezbedi ista funkcionalnost tj. razlikovanje dinamičkih tipova. U baznoj klasi se može definisati enumeracija svih tipova: *enum FormulaType { T_TRUE, T_FALSE, T_ATOM, T_NOT, T_AND, T_OR, T_IMP, T_IFF }*. Pored toga, potrebno je dodati i čisto virtuelni metod *virtual FormulaType getType() const = 0*. Svaka konkretna potklasa (potklasa koja se može instancirati) treba da implementira ovaj metod i vrati odgovarajući *enum*. Mana ove implementacije je dodavanje dodatnog metoda i potreba za menjanjem enumeracije u slučaju dodavanja novog tipa

formule. Ono što je dobro kod ovog pristupa je što je provera brža nego kada se koristi dinamička provera tipa.

Supstitucija

Jedna od podržanih operacija je zamena potformule drugom potformulom u formuli rekurzivno (metod *substitute()*). Logika ove operacije je sledeća:

1. Ukoliko je potformula koja se menja jednaka tekućoj formuli - vratiti zamensku potformulu
2. Ukoliko tekuća formula ne sadrži operande - vratiti deljeni pokazivač na tekuću formulu
3. Ukoliko tekuća formula sadrži operande - vratiti tekuću formulu sa rekurzivno izvedenom zamenom u operandima

Ako posmatramo slučaj binarnih logičkih veznika, a ima ih četiri, jedina razlika u implementaciji ove operacije je finalno sklapanje formule od izmenjenih operanada. Da bi se izbeglo dupliranje istog koda u sve četiri klase, napravljen je šablonski metod *substituteImpl<T>()* unutar klase *BinaryConnective*. Parametrizacijom ovog metoda dobija se da isti kod pokriva funkcionalnost za sve četiri klase.

Zaglavlja

atom_set.h

```
#ifndef ATOM_SET_H
#define ATOM_SET_H

#include <set>

#define UNUSED_VARIABLE(x) ((void)(x))

using AtomSet = std::set<unsigned>;

#endif // ATOM_SET_H
```

valuation.h

```
#ifndef VALUATION_H
#define VALUATION_H

#include "atom_set.h"

#include <map>
#include <iostream>

/**
* @brief The Valuation class - klasa koja predstavlja valuaciju, tj. cuva trenutnu
vrednost atoma
*/
class Valuation
{
public:
 Valuation(const AtomSet &aset = {});
```

```
 /**
  * @brief reset - cisti mapu i postavlja vrednosti svih atoma iz skupa na false
  * @param aset - skup atoma
  */
 void reset(const AtomSet &aset);
  /**
  * @brief value - getter za vrednost atoma 'p' pri trenutnoj valuaciji
  * @param p - atom cija se vrednost dohvata
  * @return vrednost atoma 'p'
  */
 bool value(unsigned p) const;
  /**
  * @brief value - setter za vrednost atoma 'p'
  * @param p - atom koji se postavlja
  * @return referencu na vrednost atoma koju treba postaviti
  */
 bool& value(unsigned p);

 /**
  * @brief next - menja vrednost valuacije na leksikografski sledecu valuaciju
  * @return true ako postoji sledeca valuacija, false inace
  */
 bool next();
  /**
  * @brief print - stampa valuaciju u stream
  * @param out - stream u koji se stampa
  * @return referencu na stream
  */
 std::ostream& print(std::ostream &out) const;
 private:
 std::map<unsigned, bool> m_values;
};

std::ostream& operator<<(std::ostream &out, const Valuation &v);

#endif // VALUATION_H
```

base_formula.h

```
#ifndef BASEFORMULA_H
#define BASEFORMULA_H

#include "atom_set.h"
#include "valuation.h"

#include <memory>
#include <iostream>
#include <experimental/optional>

/**
* Forward declaration klase BaseFormula zbog upotrebe za definisanje tipa Formula
*/
```

```
class BaseFormula;

/**
* Definisanje tip
*/
using Formula = std::shared_ptr<BaseFormula>;
using OptionalValuation = std::experimental::optional<Valuation>;

/**
* @brief The BaseFormula class - bazna klasa hijerarhije za formiranje iskaznih formula.
*/
class BaseFormula : public std::enable_shared_from_this<BaseFormula>
{
public:
 BaseFormula();
  /**
  * @brief print - stampa sadrzaj formule u C++ stream
  * @param out - stream u koji se stampa
  * @return referenca na stream (slicno kao kod operatora <<)
  */
 virtual std::ostream& print(std::ostream &out) const = 0;
  /**
  * @brief complexity - racuna slozenost formule koja je jednaka broju logickih veznika
koji formula sadrzi
  * @return slozenost formule
  */
 virtual unsigned complexity() const = 0;
  /**
  * @brief substitute - menja u tekucoj formuli sva pojavljivanja potformule 'what' sa
'with'
  * @param what - potformula koja se menja
  * @param with - potformula kojom se menja
  * @return modifikovana formula
  */
 virtual Formula substitute(const Formula &what, const Formula &with) const = 0;
  /**
  * @brief eval - racuna istinitosnu vrednost formule za datu valuaciju 'val'
  * @param val - tekuca valuacija
  * @return true ukoliko je 'val' model za tekucu formulu, false inace
  */
 virtual bool eval(const Valuation &val) const = 0;
  /**
  * @brief getAtoms - dodaje sve atome koji se javljaju u okviru formule u skup atoma
'aset'
  * @param aset - skup atoma u koji se dodaje
  */
 virtual void getAtoms(AtomSet &aset) const;
  /**
  * @brief equalTo - proverava da li su dve formule sintaksno jednake
  * @param f - formula sa kojom se poredi tekuca formula
  * @return true ako su formule sintaksno jednake, false inace
```

```
  */
 virtual bool equalTo(const Formula &f) const;
  /**
  * @brief ~BaseFormula - podsecanje, kad god je klasa bazna destruktor treba da bude
virtualan
  */
 virtual ~BaseFormula();
  /**
  * @brief isSatisfiable - ispituje da li je formula zadovoljiva
  * @return valuacija koja je model za formulu ili nista
  */
 OptionalValuation isSatisfiable() const;

  /**
  * @brief isNotTautology - ispituje da li je formula tautologija
  * @return valuacija koja nije model za fomulu ili nista
  */
 OptionalValuation isNotTautology() const;
  /**
  * @brief isConsequence - ispituje da li je prosledjena formula 'f' posledica tekuce
formule
  * @param f - formula za koju se ispituje da li je posledica
  * @return true ako je 'f' posledica tekuce formule, false inace
  */
 bool isConsequence(const Formula &f) const;
  /**
  * @brief isEquivalent - ispituje logicku jednakost formula
  * @param f - formula za koju se ispituje logicka jednakost sa tekucom formulom
  * @return true ako su formule logicki ekvivalentne, false inace
  */
 bool isEquivalent(const Formula &f) const;
  /**
  * @brief printTruthTable - ispisuje istinitosnu tablicu u C++ stream
  * @param out - stream u koji se ispisuje tablica
  */
 void printTruthTable(std::ostream &out) const;
};

/**
* @brief operator == ispituje sintaksnu jednakost dve formule
*
* @details Implementacija se svodi na pozivanje virtuelnog equalsTo() metoda.
*
* @param lhs - prva formula
* @param rhs - druga formula
* @return true ako su formule sintaksno jednake, false inace
*/
bool operator==(const Formula &lhs, const Formula &rhs);

/**
```

```
* @brief operator != ispituje sintaksnu razlicitost dve formule, svodi se na !(lhs ==
rhs)
* @param lhs
* @param rhs
* @return
*/
bool operator!=(const Formula &lhs, const Formula &rhs);

/**
* @brief operator << sluzi za ispis u C++ stream
* @param out - stream u koji se pise
* @param f - formula koja se pise u stream
* @return  referenca na stream kako bi pozivi '<<' mogli da se ulancaju
*/
std::ostream& operator<<(std::ostream &out, const Formula &f);

#endif // BASEFORMULA_H
```

atomic_formula.h

```
#ifndef ATOMICFORMULA_H
#define ATOMICFORMULA_H

#include "base_formula.h"

class AtomicFormula : public BaseFormula
{
public:
 AtomicFormula();
  virtual unsigned complexity() const;
  virtual Formula substitute(const Formula &what, const Formula &with) const;
};

#endif // ATOMICFORMULA_H
```

binary_connective.h

```
#ifndef BINARYCONNECTIVE_H
#define BINARYCONNECTIVE_H

#include "base_formula.h"

#include <string>

class BinaryConnective : public BaseFormula
{
public:
 BinaryConnective(const Formula &op1, const Formula &op2);
  virtual std::string symbol() const = 0;
  virtual std::ostream& print(std::ostream &out) const;
  virtual unsigned complexity() const;
  virtual void getAtoms(AtomSet &aset) const;
```

```
  virtual bool equalTo(const Formula &f) const;
  std::pair<Formula, Formula> operands() const;
 protected:
  template <typename Derived>
  Formula substituteImpl(const Formula &what, const Formula &with) const;
private:
 Formula m_op1;
 Formula m_op2;
};

template <typename Derived>
Formula BinaryConnective::substituteImpl(const Formula &what, const Formula &with) const
{
 if (equalTo(what))
 {
   return with;
 }
  Formula op1, op2;
 std::tie(op1, op2) = operands();
 return std::make_shared<Derived>(op1->substitute(what, with), op2->substitute(what,
with));
}

#endif // BINARYCONNECTIVE_H
```

unary_connective.h

```
#ifndef UNARYCONNECTIVE_H
#define UNARYCONNECTIVE_H

#include "base_formula.h"

class UnaryConnective : public BaseFormula
{
public:
  UnaryConnective(const Formula &op);
  unsigned complexity() const;
  void getAtoms(AtomSet &aset) const;
  bool equalTo(const Formula &f) const;
  Formula operand() const;
private:
 Formula m_op;
};

#endif // UNARYCONNECTIVE_H
```

constant.h

```
#ifndef CONSTANT_H
#define CONSTANT_H

#include "atomic_formula.h"
```

```
class True : public AtomicFormula
{
public:
 True();
  virtual std::ostream& print(std::ostream &out) const;
  virtual bool eval(const Valuation &val) const;
};

class False : public AtomicFormula
{
public:
 False();
  virtual std::ostream& print(std::ostream &out) const;
  virtual bool eval(const Valuation &val) const;
};

#endif // CONSTANT_H
```

atom.h

```
#ifndef ATOM_H
#define ATOM_H

#include "atomic_formula.h"

class Atom : public AtomicFormula
{
public:
 Atom(unsigned p);
  virtual std::ostream& print(std::ostream &out) const;
  virtual bool eval(const Valuation &val) const;
  virtual void getAtoms(AtomSet &aset) const;
  virtual bool equalTo(const Formula &f) const;
 private:
 unsigned m_p;
};

#endif // ATOM_H
```

not.h

```
#ifndef NOT_H
#define NOT_H

#include "unary_connective.h"

class Not : public UnaryConnective
{
public:
 Not(const Formula &op);
  virtual std::ostream& print(std::ostream &out) const;
```

```
  virtual Formula substitute(const Formula &what, const Formula &with) const;
  virtual bool eval(const Valuation &val) const;
};

#endif // NOT_H
```

and.h

```
#ifndef AND_H
#define AND_H

#include "binary_connective.h"

class And : public BinaryConnective
{
public:
 And(const Formula &op1, const Formula &op2);
  virtual std::string symbol() const;
  virtual Formula substitute(const Formula &what, const Formula &with) const;
  virtual bool eval(const Valuation &val) const;
};

#endif // AND_H
```

or.h

```
#ifndef OR_H
#define OR_H

#include "binary_connective.h"

class Or : public BinaryConnective
{
public:
 Or(const Formula &op1, const Formula &op2);
  virtual std::string symbol() const;
  virtual Formula substitute(const Formula &what, const Formula &with) const;
  virtual bool eval(const Valuation &val) const;
};

#endif // OR_H
```

imp.h

```
#ifndef IMP_H
#define IMP_H

#include "binary_connective.h"

class Imp : public BinaryConnective
{
public:
```

```
 Imp(const Formula &op1, const Formula &op2);
  virtual std::string symbol() const;
  virtual Formula substitute(const Formula &what, const Formula &with) const;
  virtual bool eval(const Valuation &val) const;
};

#endif // IMP_H
```

iff.h

```
#ifndef IFF_H
#define IFF_H

#include "binary_connective.h"

class Iff : public BinaryConnective
{
public:
 Iff(const Formula &op1, const Formula &op2);
  virtual std::string symbol() const;
  virtual Formula substitute(const Formula &what, const Formula &with) const;
  virtual bool eval(const Valuation &val) const;
};

#endif // IFF_H
```

Izvorni kodovi

valuation.cpp

```
#include "valuation.h"

#include <iterator>
#include <algorithm>

Valuation::Valuation(const AtomSet &aset)
{
 reset(aset);
}

void Valuation::reset(const AtomSet &aset)
{
 m_values.clear();
 for (unsigned atom : aset)
 {
   m_values[atom] = false;
 }
}

bool Valuation::value(unsigned p) const
{
```

```
 return m_values.at(p);
}

bool &Valuation::value(unsigned p)
{
 return m_values.at(p);
}

bool Valuation::next()
{
 auto first = m_values.rbegin();
 auto last = m_values.rend();
 while (first != last)
 {
   first->second = !first->second;
   if (first->second)
   {
     return true;
   }
   ++first;
 }
  return false;
}

std::ostream &Valuation::print(std::ostream &out) const
{
 for (const auto &atomValue : m_values)
 {
   out << atomValue.second << " ";
 }
 return out;
}


std::ostream &operator<<(std::ostream &out, const Valuation &v)
{
 return v.print(out);
}
```

base_formula.cpp

```
#include "base_formula.h"

#include <typeinfo>

BaseFormula::BaseFormula()
{
 }

void BaseFormula::getAtoms(AtomSet &aset) const
{
```

```
 UNUSED_VARIABLE(aset);
}

bool BaseFormula::equalTo(const Formula &f) const
{
 const BaseFormula *base = f.get();
 return typeid (*this) == typeid (*base);
}

BaseFormula::~BaseFormula()
{
 }

OptionalValuation BaseFormula::isSatisfiable() const
{
 AtomSet aset;
 getAtoms(aset);
 Valuation val{aset};
 do {
   if (eval(val))
   {
     return val;
   }
 } while (val.next());
  return {};
}

OptionalValuation BaseFormula::isNotTautology() const
{
 AtomSet aset;
 getAtoms(aset);
 Valuation val{aset};
 do {
   if (!eval(val))
   {
     return val;
   }
 } while (val.next());
 return {};
}

bool BaseFormula::isEquivalent(const Formula &f) const
{
 AtomSet aset;
 getAtoms(aset);
 f->getAtoms(aset);
 Valuation val{aset};
 do {
   if (eval(val) != f->eval(val))
   {
     return false;
```

```
    }
  } while (val.next());
  return true;
}

void BaseFormula::printTruthTable(std::ostream &out) const
{
  AtomSet aset;
  getAtoms(aset);
  Valuation v{aset};
  out << std::noboolalpha;
  do {
    out << v << " | " << eval(v) << '\n';
  } while (v.next());
}

bool BaseFormula::isConsequence(const Formula &f) const
{
  AtomSet aset;
  getAtoms(aset);
  f->getAtoms(aset);
  Valuation val{aset};
  do {
    if (eval(val) && !f->eval(val))
    {
      return false;
    }
  } while (val.next());
  return true;
}

bool operator==(const Formula &lhs, const Formula &rhs)
{
  return lhs->equalTo(rhs);
}

bool operator!=(const Formula &lhs, const Formula &rhs)
{
  return !(lhs == rhs);
}

std::ostream &operator<<(std::ostream &out, const Formula &f)
{
  return f->print(out);
}
```

atomic_formula.cpp

```
#include "atomic_formula.h"

AtomicFormula::AtomicFormula()
```

```
 : BaseFormula ()
{
 }

unsigned AtomicFormula::complexity() const
{
 return 0;
}

Formula AtomicFormula::substitute(const Formula &what, const Formula &with) const
{
 if (equalTo(what))
 {
   return with;
 }
  return std::const_pointer_cast<BaseFormula>(shared_from_this());
}
```

binary_connective.cpp

```
#include "binary_connective.h"

BinaryConnective::BinaryConnective(const Formula &op1, const Formula &op2)
 : BaseFormula (), m_op1(op1), m_op2(op2)
{
 }

std::ostream &BinaryConnective::print(std::ostream &out) const
{
 out << "(";
 m_op1->print(out);
 out << symbol();
 m_op2->print(out);
 return out << ")";
}

unsigned BinaryConnective::complexity() const
{
 return 1 + m_op1->complexity() + m_op2->complexity();
}

void BinaryConnective::getAtoms(AtomSet &aset) const
{
 m_op1->getAtoms(aset);
 m_op2->getAtoms(aset);
}

bool BinaryConnective::equalTo(const Formula &f) const
{
 if (BaseFormula::equalTo(f))
 {
```

```
   const BinaryConnective *pF = static_cast<const BinaryConnective*>(f.get());
   return m_op1 == pF->m_op1 && m_op2 == pF->m_op2;
 }
  return false;
}

std::pair<Formula, Formula> BinaryConnective::operands() const
{
 return {m_op1, m_op2};
}
```

unary_connective.cpp

```
#include "unary_connective.h"

UnaryConnective::UnaryConnective(const Formula &op)
 :BaseFormula (), m_op{op}
{
}

unsigned UnaryConnective::complexity() const
{
 return m_op->complexity() + 1;
}

void UnaryConnective::getAtoms(AtomSet &aset) const
{
 m_op ->getAtoms(aset);
}

bool UnaryConnective::equalTo(const Formula &f) const
{
 if (BaseFormula::equalTo(f))
 {
   const UnaryConnective *pF = static_cast<const UnaryConnective *>(f.get());
   return pF->m_op == m_op;
 }
  return false;
}

Formula UnaryConnective::operand() const
{
 return m_op;
}
```

constant.cpp

```
#include "constant.h"
```

```
True::True()
 : AtomicFormula ()
{
 }

std::ostream &True::print(std::ostream &out) const
{
 return out << "TRUE";
}

bool True::eval(const Valuation &val) const
{
 UNUSED_VARIABLE(val);
 return true;
}

False::False()
 : AtomicFormula ()
{
 }

std::ostream &False::print(std::ostream &out) const
{
 return out << "FALSE";
}

bool False::eval(const Valuation &val) const
{
 UNUSED_VARIABLE(val);
 return false;
}
```

atom.cpp

```
#include "atom.h"

Atom::Atom(unsigned p)
 : AtomicFormula (), m_p(p)
{
 }

std::ostream &Atom::print(std::ostream &out) const
{
 return out << "p" << m_p;
}

bool Atom::eval(const Valuation &val) const
{
 return val.value(m_p);
}
```

```
void Atom::getAtoms(AtomSet &aset) const
{
 aset.insert(m_p);
}

bool Atom::equalTo(const Formula &f) const
{
 if (BaseFormula::equalTo(f))
 {
   const Atom *pF = static_cast<const Atom*>(f.get());
   return pF->m_p == m_p;
 }
  return false;
}
```

not.cpp

```
#include "not.h"

Not::Not(const Formula &op)
 : UnaryConnective (op)
{
 }

std::ostream &Not::print(std::ostream &out) const
{
 out << "~";
 return operand()->print(out);
}

Formula Not::substitute(const Formula &what, const Formula &with) const
{
 if (equalTo(what))
 {
   return with;
 }
  return std::make_shared<Not>(operand()->substitute(what, with));
}

bool Not::eval(const Valuation &val) const
{
 return !operand()->eval(val);
}
```

and.cpp

```
#include "and.h"

And::And(const Formula &op1, const Formula &op2)
 :BinaryConnective (op1, op2)
```

```
{
 }

std::string And::symbol() const
{
 return "/\\";
}

Formula And::substitute(const Formula &what, const Formula &with) const
{
 return substituteImpl<And>(what, with);
}

bool And::eval(const Valuation &val) const
{
 Formula op1, op2;
 std::tie(op1, op2) = operands();
  return op1->eval(val) && op2->eval(val);
}
```

or.cpp

```
#include "or.h"

Or::Or(const Formula &op1, const Formula &op2)
 : BinaryConnective (op1, op2)
{
 }

std::string Or::symbol() const
{
 return "\\/";
}

Formula Or::substitute(const Formula &what, const Formula &with) const
{
 return substituteImpl<Or>(what, with);
}

bool Or::eval(const Valuation &val) const
{
 Formula op1, op2;
 std::tie(op1, op2) = operands();
  return op1->eval(val) || op2->eval(val);
}
```

imp.cpp

```
#include "imp.h"
```

```
Imp::Imp(const Formula &op1, const Formula &op2)
 :BinaryConnective (op1, op2)
{
 }

std::string Imp::symbol() const
{
 return "=>";
}

Formula Imp::substitute(const Formula &what, const Formula &with) const
{
 return substituteImpl<Imp>(what, with);
}

bool Imp::eval(const Valuation &val) const
{
 Formula op1, op2;
 std::tie(op1, op2) = operands();
  return !op1->eval(val) || op2->eval(val);
}
```

iff.cpp

```
#include "iff.h"

Iff::Iff(const Formula &op1, const Formula &op2)
 :BinaryConnective (op1, op2)
{
 }

std::string Iff::symbol() const
{
 return "<=>";
}

Formula Iff::substitute(const Formula &what, const Formula &with) const
{
 return substituteImpl<Iff>(what, with);
}

bool Iff::eval(const Valuation &val) const
{
 Formula op1, op2;
 std::tie(op1, op2) = operands();
  return op1->eval(val) == op2->eval(val);
}
```

Čas 2 - iskazna logika, pojednostavljivanje formula, normalne forme

Preduslovi: *čas 1; predavanja - teorema o zameni;*

SAT problem i uvod u normalne forme

SAT problem predstavlja problem zadovoljivosti formula iskazne logike. Sa strane teorijskog značaja to je prvi problem za koji je pokazano da je *NP kompletan*. S praktične strane, za koju je lepo primetiti da sledi iz teorijske, SAT problem je važan jer za njega postoje efikasne procedure rešavanja, što nam sa činjenicom da se drugi *NP teški* problemi mogu svesti na njega u polinomijalnom vremenu, daje efikasnu proceduru za rešavanje čitavog niza problema. Da bi se efikasno rešio novi *NP težak* problem jednostavno ga treba svesti na SAT problem.
Posmatrajmo sada jedan ilustrativni primer koji će nas dovesti do motivacije zašto ćemo koristiti normalne forme u radu sa formulama iskazne logike. Neka je dat isti polinom u dva oblika (dve forme):

- x^2 - 3x + 3
- (x-1)(x-2)

Ako je zadatak pronaći nule polinoma jasno se vidi da nam je druga reprezentacija mnogo zgodnija za rad (nule se trivijalno prepoznaju kao 1 i 2). Generalno, postupak trazenja nula za ovakve jednostavne polinome mogao bi se svesti na svođenje na reprezentaciju u kojoj se množe dve zagrade kao što je prikazano. Slično, kada radimo sa formulama iskazne logike postoje neke reprezentacije koje nam omogućavaju brže rešavanje SAT problema.

Pojednostavljivanje formula

Pre nego što pređemo na konkretne reprezentacije, bilo bi dobro da pogledamo šta bismo mogli da uradimo sa formulama koji do nas stignu kako bi prilikom prevođenja u normalne forme imali što manje posla. Za početak posmatrajmo formulu:

p /\ (q \/ F) /\ T

Očigledno je da kada imamo izraz oblika *q \/ F* da *F* (netačno) ne utiče na vrednost samog izraza. Celokupna vrednost zavisi od vrednosti atoma *q*. Slično, kada imamo konjunkciju oblika *r /\ T*, vrednost izraza zavisi samo od vrednosti atoma *r*. Dakle, navedena formula se može uprostiti u oblik:

p /\ q

Prikazani primer ima svoju teorijsku potporu, ako je pravilo koje primenjujemo tautologija, onda na osnovu *teoreme o zameni* tvrdimo da je transformacija semantički ispravna. Naredne tautologije ćemo koristiti da uprostimo (zamenimo) delove iskaznih formula:

- ~T -> F ~F -> T
- Formula /\ F -> F Formula /\ T -> Formula *(važi i kada operandi zamene mesta)*
- Formula \/ T -> T Formula \/ F -> Formula *(važi i kada operandi zamene mesta)*
- Formula => F -> ~Formula F => Formula -> T Formula => T -> T T => Formula -> Formula

- Formula <=> T -> Formula Formula <=> F -> ~Formula *(važi i kada operandi zamene mesta)*

Napomena da identitet važi i kada operandi zamene mesta se odnose na to da *p /\ q* i *q /\ p* imaju istu vrednost zbog komutativnosti logičkog veznika /\. Dakle, imamo ukupno 18 ovakvih pravila (2 + 4*4).

Pre nego što formule budemo svodili na neku normalnu formu mi ćemo ih uprošćavati, zbog čega se u baznu klasu *BaseFormula* dodaje novi čisto virtuelni metod koji vraća uprošćenu formulu:

```
virtual Formula simplify() const = 0;
```

Normalne forme

U nastavku ćemo razmatrati 3 normalne forme:

- Negaciona normalna forma (u daljem tekstu NNF)
- Disjunktivna normalna forma (u daljem tekstu DNF)
- Konjunktivna normalna forma (u daljem tekstu KNF)

Pre nego što pređemo na definicije, treba se podsetiti da pojam *literal* označava atom ili njegovu negaciju. Na primer *~p* i *p* su literali (i to suprotni literali), dok *~(p /\ q)* očigledno nije literal.

Def 2.1: Formula je u NNF akko je sastavljena od literala korišćenjem isključivo veznika /\ i \/ ili je logička konstanta T odnosno F.

NNF je zapravo međukorak za dobijanje KNF ili DNF što je krajnji cilj svođenja na normalne forme. Do NNF se dolazi uklanjanjem implikacija i ekvivalencija, zatim spuštanjem negacija (*f1* i *f2* su formule):

- f1 => f2 -> ~f1 \/ f2
- ~(f1 => f2) -> f1 /\ ~f2
- f1 <=> f2 -> (~f1 \/ f2) /\ (~f2 \/ f1) ili -> (f1 /\ f2) \/ (f1 \/ f2) *(ili su oba tačna ili su oba netačna)*
- ~(f1 <=> f2) -> (f1 /\ ~f2) \/ (~f1 /\ f2) ili -> (f1 \/ f2) /\ (~f1 \/ ~f2) *(jedan je tačan a drugi ne)*
- ~(f1 /\ f2) -> ~f1 \/ ~f2
- ~(f1 \/ f2) -> ~f1 /\ ~f2
- ~~f1 -> f1

Def 2.2: Formula je u DNF ako je oblika $D_1 \vee D_2 \vee \dots \vee D_N$ pričemu je svaki D_i oblika $l_1 \wedge l_2 \wedge \dots \wedge l_{ki}$, a l su literali.

Jednostavno rečeno formula je u DNF ako je disjunkcija više konjunkcija. Možemo primetiti da za formulu u DNF skoro trivijalno možemo da odredimo zadovoljivost. Dovoljno je da podesimo vrednosti literala tako da bar jedan D_i bude tačan. Međutim problem nastupa u činjenici da se samim postupkom prevođenja u DNF može dobiti eksponencijalno veći broj disjunkata i značajno veća formula od polazne pa je složenost zapravo sakrivena u samoj transformaciji formule. Kada prevodimo formulu u NNF podvučeno pravilo za razbijanje implikacije zapravo duplira formulu na koju se primenjuje. Dodatno, ne postoji neki drugi efikasan postupak za direktno prevođenje u DNF.

Def 2.3: Formula je u KNF ako je oblika $K_1 \wedge K_2 \wedge \dots \wedge K_N$ pri čemu je svaki K_i oblika $l_1 \vee l_2 \vee \dots \vee l_{ki}$, a l su literali.

Jednostavno rečeno formula je u KNF ako je konjunkcija više disjunkcija. Pažljiv čitalac bi mogao da primeti da ako je formula u DNF ili KNF ona je automatski i u NNF. Od NNF do DNF se može stići primenom sledećih logičkih ekvivalencija:

- f1 /\ (f2 \/ f3) -> (f1 /\ f2) \/ (f1 /\ f3)
- (f1 \/ f2) /\ f3 -> (f1 /\ f3) \/ (f2 /\ f3)

Slično, od NNF do KNF se može doći primenom:

- f1 \/ (f2 /\ f3) -> (f1 \/ f2) /\ (f1 \/ f3)
- (f1 /\ f2) \/ f3 -> (f1 \/ f3) /\ (f2 \/ f3)

Kada se do KNF dolazi preko NNF takođe se dolazi do problema eksponencijalnog uvećanja formule. Međutim za KNF postoji postupak kojim se polazna formula može prevesti u KNF formulu koja je ekvizadovoljiva u odnosu na polaznu i samo za konstantan faktor veća. U pitanju je Cajtinova definiciona KNF. Sam algoritam će biti predstavljen u okviru gradiva za sledeći čas, ono što je trenutno bitno je da od polazne formule do KNF sa kojom možemo nešto da radimo možemo stići algoritmom koji nije eksponencijalan zbog čega većina SAT rešavača prihvata formule baš u KNF (iako za formulu koja je u DNF zadovoljivost ispitujemo u linearnom vremenu).

Što se naše implementacije tiče, u baznu klasu dodajemo metode koji su u skladu sa planom transformacije formule: *Formula* --> pojednostavljivanje --> NNF --> KNF --> *Formula'* pri čemu je rezultujuća formula u KNF-u (postupa je sličan i za DNF):

```
using LiteralList = std::vector<Formula>;
using LiteralListList = std::vector<LiteralList>;
class BaseFormula : public std::enable_shared_from_this<BaseFormula>
{
public:
…
 virtual Formula simplify() const = 0;
 virtual Formula nnf() const = 0;
 virtual LiteralListList listCNF() const = 0;
 virtual LiteralListList listDNF() const = 0;
};
```

LiteralList je tip koji se koristi da se predstavi jedna klauza (ili jedna konjunkcija u slučaju DNF), pri čemu treba primetiti da postoji dodatno ograničenje da je svaki element vektora zapravo atom ili njegova negacija (iako je korišćeni tip *Formula*). KNF odnosno DNF se dobija kao niz klauza, odnosno konjunkcija, pa se samim tim koristi vektor klauza (tip *LiteralListList*).

Implementacija

Detalji na koje treba obratiti pažnju

Uprošćavanje i NNF

Počnimo sa pojednostavljivanjem formula. Strategija pojednostavljivanja bi trebalo da bude jasna do sada, ali ipak pogledajmo jedan primer zbog kompletnosti. Neka je data formula *(p /\ T) \/ (q /\ F)*, rezultat pojednostavljivanja treba da bude samo konstanta T. Dakle, celokupna formula je tipa *Or* i ako pokušamo direktno da je uprostimo vidimo da nije moguće primeniti nijedno pravilo za uprošćavanje (podsetiti se pravila iz Pojednostavljivanje formula). S druge strane, pojedinačne disjunkte je jednostavno uprostiti direktnom primenom pravila, prvi disjunkt ide u T, a drugi u F. Ovakvim razmišljanjem se dolazi do sledeće rekurzivne strategije date u koracima:

1. uprostiti operande ukoliko ih ima
2. uprostiti tekuću formulu

Nakon uprošćavanja disjunkata dobija se formula *T \/ F* od koje se primenom koraka 2. dolazi do konstante T. Nakon formiranja ove strategije potrebno je još obraditi neke finese. Šta se dešava na primer sa formulom *F <=> F*? Po našoj strategiji uprostimo operande, nakon čega su oni i dalje F, zatim primenimo pravilo *Formula <=> F -> ~Formula* (svejedno je da li primenjujemo ovo ili

simetrično pravilo) i dobijamo rezultujuću formula *~F* koja očigledno nije uprošćena do kraja. Taj poslednji detalj se odnosi na to da ukoliko pravilo koje primenjujemo uključuje formiranje nove formule tipa *Not* potrebno je pozvati uprošćavanje za novonastalu formulu:

```
std::make_shared<Not>(simplifiedOp1)->simplify();
```

S obzirom na sličnu prirodu operacija za prevođenje u NNF - slična je i strategija. Potrebno je prebaciti operande u NNF primenom odgovarajućih pravila (Normalne forme), a zatim biti pažljiv prilikom pravljenja novih formula tipa *Not* da se i za njih pozove metod *nnf()*. Nema potrebe za pozivanjem *nnf()* metoda na operandom od koga će se praviti formula tipa *Not*, pozvaćemo *nnf()* direktno nad novodobijenom *Not()* formulom.

KNF i DNF

Zbog kompletnosti, pređimo još jednom korake visokog nivoa za prebacivanje ulazne formule u KNF ili DNF:

1. Uprostiti formulu (*simplify()* metod)
2. Prebaciti formulu u NNF (*nnf()* metod)
3. Prebaciti formulu u KNF ili DNF

Dakle možemo smatrati da nam je ulaz formula u NNF. Ona se sastoji samo od literala i veznika /\ odnosno \/. Pogledajmo jedan primer za prebacivanje u KNF, što se tiče DNF postupak je sličan. Neka je data formula *(p /\ q /\ ~r) \/ (s /\ t)*, njeno prebacivanje u KNF se svodi na "množenje" zagrada, dakle rezultat bi bio *(p \/ s) /\ (p \/ t) /\ (q \/ s) /\ (q \/ t) /\ (~r \/ s) /\ (~r \/ t)*. Ako bismo malo proširili polaznu formulu, recimo da ona izgleda ovako ***(x \/ y)*** */\ (p /\ q /\ ~r) \/ (s /\ t)* dobili bismo sličan rezultat ***(x \/ y \/ s) /\ (x \/ y \/ t)*** */\ (p \/ s) /\ (p \/ t) /\ (q \/ s) /\ (q \/ t) /\ (~r \/ s) /\ (~r \/ t)* (razlika je data u tekstu masnim slovima). Primećujemo da su rezultujuće klauze koje su date masnim slovima takođe dobijene operacijom "množenja" između *(x \/ y)* i *(s /\ t)*. Dakle, kada imamo operande koji su razdvojeni sa */* zaključujemo da je željeni rezultat prevođenja u KNF proizvod literala koji se u njima nalaze. S druge strane pogledajmo još jednom kako je dobijena podvučena rezultujuća formula. Imali smo dva proizvoda, *(x \/ y)* sa *(s /\ t)* i *(p /\ q /\ ~r)* sa *(s /\ t)*, i na kraju jedno spajanje dobijenih proizvoda - konkatenaciju dve liste klauza. Ove operacije ćemo koristiti da od NNF dođemo do KNF ili DNF (množenje listi i konkatenaciju listi). Dodajemo statičke metode u baznu klasu:

```
using LiteralList = std::vector<Formula>;
using LiteralListList = std::vector<LiteralList>;
class BaseFormula : public std::enable_shared_from_this<BaseFormula>
{
public:
…
 template <typename ListType>
 static ListType concatenate(const ListType &l1, const ListType &l2);
 static LiteralListList cross(const LiteralListList &l1, const LiteralListList &l2);
};
```

Poslednji detalj na koji treba obratiti pažnju se odnosi na to da imamo potrebu za dve različite konkatenacije. Jedna je da izvršimo konkatenaciju dve klauze (*LiteralList*), a druga da spojimo dve liste klauza (*LiteraListList*). Zbog toga pravimo metod *concatenate()* da bude šablonski - izbegavamo nepotrebno dupliranje koda jer isti algoritam spajanja radi u oba slučaja.

Izvorni kod metoda

Za dohvatanje operanada kod klasa koje nasleđuju *BinaryConnective* koristi se makro:

```
#define GET_OPERANDS(x, y) \
       Formula x, y; \
       std::tie(x, y) = operands();
```

Pojednostavljivanje (*simplify()*)

```
Formula AtomicFormula::simplify() const
{
 return std::const_pointer_cast<BaseFormula>(shared_from_this());
}

Formula Not::simplify() const
{
 Formula simplifiedOp = operand()->simplify();
 if (BaseFormula::isOfType<True>(simplifiedOp))
 {
   return std::make_shared<False>();
 }
 else if (BaseFormula::isOfType<False>(simplifiedOp))
 {
   return std::make_shared<True>();
 }
  return std::make_shared<Not>(simplifiedOp);
}

Formula And::simplify() const
{
 GET_OPERANDS(simplifiedOp1, simplifiedOp2);
 simplifiedOp1 = simplifiedOp1->simplify();
 simplifiedOp2 = simplifiedOp2->simplify();
  if (BaseFormula::isOfType<True>(simplifiedOp1))
 {
   return simplifiedOp2;
 }
 else if (BaseFormula::isOfType<True>(simplifiedOp2))
 {
   return simplifiedOp1;
 }
 else if (BaseFormula::isOfType<False>(simplifiedOp1))
 {
   return std::make_shared<False>();
 }
 else if (BaseFormula::isOfType<False>(simplifiedOp2))
 {
   return std::make_shared<False>();
 }
  return std::make_shared<And>(simplifiedOp1, simplifiedOp2);
}
```

```
Formula Or::simplify() const
{
 GET_OPERANDS(simplifiedOp1, simplifiedOp2);
 simplifiedOp1 = simplifiedOp1->simplify();
 simplifiedOp2 = simplifiedOp2->simplify();
  if (BaseFormula::isOfType<True>(simplifiedOp1))
 {
   return simplifiedOp1;
 }
 else if (BaseFormula::isOfType<True>(simplifiedOp2))
 {
   return simplifiedOp2;
 }
 else if (BaseFormula::isOfType<False>(simplifiedOp1))
 {
   return simplifiedOp2;
 }
 else if (BaseFormula::isOfType<False>(simplifiedOp2))
 {
   return simplifiedOp1;
 }
  return std::make_shared<Or>(simplifiedOp1, simplifiedOp2);
}

Formula Imp::simplify() const
{
 GET_OPERANDS(simplifiedOp1, simplifiedOp2);
 simplifiedOp1 = simplifiedOp1->simplify();
 simplifiedOp2 = simplifiedOp2->simplify();
  if (BaseFormula::isOfType<True>(simplifiedOp1))
 {
   return simplifiedOp2;
 }
 else if (BaseFormula::isOfType<True>(simplifiedOp2))
 {
   return simplifiedOp1;
 }
 else if (BaseFormula::isOfType<False>(simplifiedOp1))
 {
   return std::make_shared<True>();
 }
 else if (BaseFormula::isOfType<False>(simplifiedOp2))
 {
   return std::make_shared<Not>(simplifiedOp1)->simplify();
 }
  return std::make_shared<Imp>(simplifiedOp1, simplifiedOp2);
}

Formula Iff::simplify() const
{
```

```
 GET_OPERANDS(simplifiedOp1, simplifiedOp2);
 simplifiedOp1 = simplifiedOp1->simplify();
 simplifiedOp2 = simplifiedOp2->simplify();
  if (BaseFormula::isOfType<True>(simplifiedOp1))
 {
   return simplifiedOp2;
 }
 else if (BaseFormula::isOfType<True>(simplifiedOp2))
 {
   return simplifiedOp1;
 }
 else if (BaseFormula::isOfType<False>(simplifiedOp1))
 {
   return std::make_shared<Not>(simplifiedOp2)->simplify();
 }
 else if (BaseFormula::isOfType<False>(simplifiedOp2))
 {
   return std::make_shared<Not>(simplifiedOp1)->simplify();
 }
  return std::make_shared<Iff>(simplifiedOp1, simplifiedOp2);
}
```

Transformacija u NNF

```
Formula AtomicFormula::nnf() const
{
 return std::const_pointer_cast<BaseFormula>(shared_from_this());
}

Formula Not::nnf() const
{
 Formula op = operand();
 if (BaseFormula::isOfType<Not>(op))
 {
   const Not *opNot = static_cast<const Not*>(op.get());
   return opNot->operand()->nnf();
 }
 else if (BaseFormula::isOfType<And>(op))
 {
   const And *opAnd = static_cast<const And*>(op.get());
   Formula op1, op2;
   std::tie(op1, op2) = opAnd->operands();
   return std::make_shared<Or>(std::make_shared<Not>(op1)->nnf(),
std::make_shared<Not>(op2)->nnf());
 }
 else if (BaseFormula::isOfType<Or>(op))
 {
   const Or *opOr = static_cast<const Or*>(op.get());
   Formula op1, op2;
   std::tie(op1, op2) = opOr->operands();
   return std::make_shared<And>(std::make_shared<Not>(op1)->nnf(),
std::make_shared<Not>(op2)->nnf());
```

```
 }
 else if (BaseFormula::isOfType<Imp>(op))
 {
   const Imp *opImp = static_cast<const Imp*>(op.get());
   Formula op1, op2;
   std::tie(op1, op2) = opImp->operands();
   return std::make_shared<And>(op1->nnf(), std::make_shared<Not>(op2)->nnf());
 }
 else if (BaseFormula::isOfType<Iff>(op)) /* ovde dolazi do dupliranje formule */
 {
   const Iff *opIff = static_cast<const Iff*>(op.get());
   Formula op1, op2;
   std::tie(op1, op2) = opIff->operands();
   return std::make_shared<Or>(std::make_shared<And>(op1->nnf(),
std::make_shared<Not>(op2)->nnf()),
                               std::make_shared<And>(std::make_shared<Not>(op1)->nnf(),
op2->nnf()));
 }
  return std::const_pointer_cast<BaseFormula>(shared_from_this());
}

Formula And::nnf() const
{
 GET_OPERANDS(op1, op2);
 return std::make_shared<And>(op1->nnf(), op2->nnf());
}

Formula Or::nnf() const
{
 GET_OPERANDS(op1, op2);
 return std::make_shared<Or>(op1->nnf(), op2->nnf());
}

Formula Imp::nnf() const
{
 GET_OPERANDS(op1, op2);
 return std::make_shared<Or>(std::make_shared<Not>(op1)->nnf(), op2->nnf());
}

Formula Iff::nnf() const
{
 GET_OPERANDS(op1, op2);
  return std::make_shared<And>(std::make_shared<Or>(std::make_shared<Not>(op1)->nnf(),
op2->nnf()),
                              std::make_shared<Or>(op1->nnf(),
std::make_shared<Not>(op2)->nnf()));
}
```

Operacije sa klauzama i listama klauza

```
template <typename ListType>
ListType BaseFormula::concatenate(const ListType &l1, const ListType &l2)
```

```
{
 ListType result;
 result.reserve(l1.size() + l2.size());
 std::copy(l1.cbegin(), l1.cend(), std::back_inserter(result));
 std::copy(l2.cbegin(), l2.cend(), std::back_inserter(result));
 return result;
}

LiteralListList BaseFormula::cross(const LiteralListList &l1, const LiteralListList &l2)
{
 LiteralListList result;
 result.reserve(l1.size() * l2.size());
 for (const auto& outerL : l1)
 {
   for (const auto& innerL : l2)
   {
     result.push_back(concatenate(outerL, innerL));
   }
 }
  return result;
}
```

Prebacivanje u KNF i DNF

```
LiteralListList True::listCNF() const
{
   /*
    * Klauze su ogranicenja - prazna lista klauza je uvek zadovoljiva
    * tj. (nema nezadovoljenih ogranicenja) - ovo je konvencija.
    */
   return {};
}

LiteralListList True::listDNF() const
{
   /*
    * DNF je tacna ako ima bar jednu listu literala koja je zadovoljena,
    * a lista je konjukncija za koju vazi da je zadovoljena jer nema
    * nijedan nezadovoljeni literal - ovo je konvencija.
    */
   return {{}};
}
LiteralListList False::listCNF() const
{
   /*
    * Prazna klauza je uvek nezadovoljena (potsetiti se rezolucije) - ovo je
    * konvencija.
    */
   return {{}};
}

LiteralListList False::listDNF() const
```

```
{
   /*
    * Prazna lista listi literala je nezadovoljena jer ne postoji nijedna
    * lista literala koja je zadovoljena - ovo je konvencija.
    */
   return {};
}

LiteralListList Atom::listCNF() const
{
 return {{std::const_pointer_cast<BaseFormula>(shared_from_this())}};
}

LiteralListList Atom::listDNF() const
{
 return {{std::const_pointer_cast<BaseFormula>(shared_from_this())}};
}

LiteralListList Not::listCNF() const
{
 return {{std::const_pointer_cast<BaseFormula>(shared_from_this())}};
}

LiteralListList Not::listDNF() const
{
 return {{std::const_pointer_cast<BaseFormula>(shared_from_this())}};
}

LiteralListList And::listCNF() const
{
 GET_OPERANDS(op1, op2);
 return BaseFormula::concatenate(op1->listCNF(), op2->listCNF());
}

LiteralListList And::listDNF() const
{
 GET_OPERANDS(op1, op2);
 return BaseFormula::cross(op1->listDNF(), op2->listDNF());
}

LiteralListList Or::listCNF() const
{
 GET_OPERANDS(op1, op2);
 return BaseFormula::cross(op1->listCNF(), op2->listCNF());
}

LiteralListList Or::listDNF() const
{
 GET_OPERANDS(op1, op2);
 return BaseFormula::concatenate(op1->listDNF(), op2->listDNF());
}
```

```
LiteralListList Imp::listCNF() const
{
 throw std::runtime_error{"Implikacija se mora eliminisati tokom nnf() procedure"};
}

LiteralListList Imp::listDNF() const
{
 throw std::runtime_error{"Implikacija se mora eliminisati tokom nnf() procedure"};
}

LiteralListList Iff::listCNF() const
{
 throw std::runtime_error{"Ekvivalencija se mora eliminisati tokom nnf() procedure"};
}

LiteralListList Iff::listDNF() const
{
 throw std::runtime_error{"Ekvivalencija se mora eliminisati tokom nnf() procedure"};
}
```

Provera tipa formule

```
template <typename Derived>
const Derived* BaseFormula::isOfType(const Formula &f)
{
 return dynamic_cast<const Derived *>(f.get());
}
```

Čas 3 - Cajtinova transformacija, DPLL procedura

Preduslovi: *čas 1 i čas 2; rezolucija u iskaznoj logici (predmet Veštačka Inteligencija);*

Cajtinova transformacija

U prethodnoj sekciji je objašnjen značaj normalnih formi za izraze uopšte, a naglasak je stavljen posebno na izraze iskazne logike - logičke formule. Problem koji je predstavljen se odnosio na eksponencijalno uvećanje formula prilikom transformacije u KNF odnosno DNF. Cajtinova transformacija nam omogućava da proizvoljnu formulu *Formula* prebacimo u formulu *Formula'* koja je u KNF, pri čemu važi da je svaki model za *Formula'* ujedno i model za *Formula*, dok se svaki model za *Formula* može proširiti do modela za *Formula'*. Dobijena KNF formula je najviše za konstantan faktor veća od polazne. Složenost transfromacije je linearna, a ne eksponencijalna.
Ideja transformacije je jednostavna, za svaku potformulu *q* uvodi se novo iskazno slovo s_i kojim se ona menja uz dodavanje novog konjunkta s_i <=> *q* u početnu formulu. Pogledajmo mali primer:

(p => (q /\ r)) => (r <=> (q \/ ~p)), uvodimo s_1 <=> (q /\ r), s_2 <=> (q \/ ~p)

(p => s_1) => (r <=> s_2) /\ s_1 <=> (q /\ r) /\ s_2 <=> (q \/ ~p), uvodimo s_3 <=> (p => s_1), s_4 <=> (r <=> s_2)

(s_3 => s_4) **/** (s_3 <=> (p => s_1)) **/** (s_4 <=> (r <=> s_2)) **/** (s_1 <=> (q /\ r)) **/** (s_2 <=> (q \/ ~p))

Podvučeni delovi se odnose na nove konjunkte dodate nakon odgovarajućih zamena potformula novim atomima. Ovako dobijena formula se može pojednostaviti tako da se dobije:
$(\sim s_3 \vee s_4) \wedge (\sim s_1 \vee q) \wedge (\sim s_1 \vee r) \wedge (\sim q \vee \sim r \vee s_1) \wedge$
$(\sim s_1 \vee q \vee \sim p) \wedge (\sim q \vee s_2) \wedge (p \vee s_2) \wedge (\sim s_3 \vee \sim p \vee s_1) \wedge$
$(p \vee s_3) \wedge (\sim s_1 \vee s_3) \wedge (\sim s_4 \vee \sim r \vee s_2) \wedge (\sim s_4 \vee r \vee \sim s_2) \wedge$
$(r \vee s_2 \vee s_4) \wedge (\sim s_2 \vee \sim r \vee s_4)$
Prikazaćemo razlaganje $s_1 <=> (q \wedge r)$, a na čitaocu je da primeni slično razlaganje na ostale ekvivalencije. Pravilom $p <=> q \rightarrow (\sim p \vee q) \wedge (p \vee \sim q)$ dolazimo do (podsetiti se pravila iz sekcije Normalne forme):

$$(\sim s_1 \vee (q \wedge r)) \wedge (s_1 \vee \sim(q \wedge r))$$
$$(\sim s_1 \vee q) \wedge (\sim s_1 \vee r) \wedge (s_1 \vee \sim q \vee \sim r)$$

Vidmo da smo od dva početna binarna veznika (<=> i /\), dobili šest binarnih veznika na ovom primeru. Dakle, postavlja se pitanje šta se tačno dešava sa veličinom formule, kakvu garanciju imamo da formula neće eksponencijalno narasti? Odgovor leži u potformulama. Naime, kada smo naivnim metodom prevodili formulu u KNF imali smo problem sa prevođenjem ekvivalencija jer se baš tada formula duplirala i to kada se desi da je ekvivalencija najviši veznik u stablu izraza. Ovde imamo drugačiju situaciju, uvodimo implikacije za male potformule i samo se one mogu duplirati. Kako je broj potformula ograničen brojem logičkih veznika, odatle vidimo da će se formula uvećati najviše za konstantan faktor (najviše *3n+2* puta, gde je *n* broj logičkih veznika početne formule).

DPLL procedura

DPLL algoritam je skraćenica za Davis–Putnam–Logemann–Loveland algoritam. Predstavlja osnovu za najsofisticiranije SAT rešavače danas, koji doduše implementiraju veći broj optimizacija u odnosu na originalno objavljen algoritam (**C**onflict-**D**riven-**C**lause-**L**earning to jest CDCL rešavači).

Osnovni algoritam

Naivan metod za proveru zadovoljivosti bi mogao biti izlistavanje svih permutacija vrednosti atoma iskazne formule - praktično generisanje istinitosne tablice. Ovu ideju možemo malo poboljšati, zašto nastavljati sa narednim izborima ako izabrane vrednosti atoma već ne zadovoljavaju neku od klauza? Ovaj pristup je malo bolji od slepog generisanja svih valuacija. S druge strane, prilikom takvog algoritma, u svakom trenutku se donosi odluka “hajde da postavimo literal *l* na vrednost T” ili “hajde da postavimo literal *l* na vrednost F”. Ove odluke se donose nezavisno od toga da li ima smisla postaviti literal *l* na odgovarajuću vrednost ili ne (slučajnim biranjem). Ispostavlja se da postoje situacije kada ovakva odluka ne mora da se donosi slučajno. Pre nego što se pozabavimo “pravim” izborom literala, možemo formirati jednostavan algoritam:

1. Ako je skup klauza prazan - vrati SAT
2. Ako skup klauza sadrži praznu klauzu - vrati UNSAT
3. U suprotnom
    a. Pronađi literal *l* koji se pojavljuje u *Formula*
    b. Ako je formula *Formula*[*l*->T] zadovoljiva vrati SAT (*l* se menja sa T u formuli)
    c. Inače vrati da li je *Formula*[*l*->F] zadovoljiva (*l* se menja sa F u formuli)

Što se tiče navedene procedure i skupa klauza, ukoliko *l*->T, uklonimo sve klauze koje sadrže *l* jer su one sigurno zadovoljene. Iz svih klauza koje sadrže *~l* uklonimo *~l* jer taj literal koji je F sigurno ne može da zadovolji pomenute klauze. Predstavljena procedura je rekurzivna i podrazumeva da se

različita stanja formule pamte na steku što je memorijski jako neefikasno. Da bismo prevazišli ovaj problem uvodimo drugi način da pamtimo stanje formule u nekom trenutku - *parcijalnu valuaciju*.

Def 3.1: Parcijalna valuacija je skup koji ne sadrži dva suprotna literala. Kažemo da je literal tačan u parcijalnoj valuaciji ako se nalazi u skupu, a nedefinisan inače.

Korišćenjem parcijalne valuacije menja se značenje prazne klauze (konfliktne klauze). Klauza je prazna ako parcijalna valuacija sadrži suprotan literal za svaki literal te klauze. S druge strane, klauza je zadovoljena ako se svi njeni literali nalaze u parcijalnoj valuaciji.

Dve stvari je potrebno specijalno naglasiti. Upotrebom parcijalne valuacije izbegavamo potrebu za modifikacijom formule i rešavamo memorijski problem. S druge strane parcijalna valuacija nam ne omogućava jednostavnu proveru da li je formula tačna u nekoj parcijalnoj valuaciji. Međutim, ukoliko smo došli do toga da parcijalna valuacija sadrži vrednost za sve atome koji se javljaju u formuli i da ne postoji konfliktna klauza, tada slobodno možemo da zaključimo da je formula tačna u toj parcijalnoj valuaciji.

Elementi zaključivanja

Vratimo se na izbor literala čiju ćemo vrednost postaviti, pogledajmo mali primer (PVal je parcijalna valuacija):

(p \/ q) /\ (~p \/ r \/ s) /\ (t \/ ~s \/ r \/ ~q), PVal=[~p]

Podvučena klauza je specijalna zbog toga što parcijalna valuacija sadrži *~p* što praktično svodi klauzu na *(F \/ q) -> q*. Dakle, ako se desi da u parcijalnu valuaciju dodamo *~q* sigurno će postojati konfliktna klauza (baš ta podvučena). Odatle možemo da zaključimo da u parcijalnu valuaciju u kojoj se nalazi *~p* moramo dodati *q*. Klauze za koje parcijalna valuacija sadrži sve suprotne literale osim jednog koji je nedefinisan zovu se jedinične klauze. Za ovakve klauze sa sigurnošću možemo da tvrdimo da smemo da propagiramo nedefinisani literal u parcijalnu valuaciju.

Pogledajmo isti primer još jednom:

(p \/ q) /\ (~p \/ r \/ s) /\ (t \/ ~s \/ r \/ ~q), PVal=[~p]

Ono što je zanimljivo za podvučeni literal *r* je to da se u celoj formuli ne nalazi njemu suprotan literal *~r*. To znači da mi možemo sa sigurnošću da tvrdimo da ako postoji model za formulu čiju zadovoljivost ispitujemo, onda taj model mora da sadrži literal *r*.

Dva navedena poboljšanja su u engleskoj literaturi poznata kao *unit propagation* (jedinična propagacija u nastavku) i *pure literal* (jedinstveni literal u nastavku). Ispostavlja se da je jedinična propagacija korisnije poboljšanje i za ovu tehniku postoji jako efikasna implementacija - šema posmatranja dva literala (pogledati za Napredna poboljšanja opis procedure). S druge strane jedinstveni literal se najčešće koristi kao tehnika za preprocesiranje formule. Jednom kada pretraga krene, traženje jedinstvenih literala povećava broj potrebnih operacija više nego što pronalazak ovakvih literala ubrzava pretragu. Novi algoritam bismo mogli da sumiramo u par koraka:

1. Pre pretrage izbaciti iz skupa klauza sve klauze koje sadrže neki jedinstveni literal
2. Pretraga:
    a. Ako smo naišli na konfliktnu klauzu vrati UNSAT
    b. Inače, ako je parcijalna valuacija potpuna vrati SAT
    c. Inače
        i. Ako može da se uradi jedinična propagacija dodaj literal *l* u parcijalnu valuaciju i nastavi pretragu
        ii. Inače, izaberi nedefinisani literal *l*, dodaj ga u parcijalnu valuaciju i nastavi pretragu

Iterativna implementacija

Posmatrajmo malo šta se dešava sa pretragom sada kada smo uveli propagiranje jediničnih klauza. Pogledajmo primer (precrtane klauze su zadovoljene, masni literali u PVal su nasumično odabrani, dok su ostali dobijeni jediničnom propagacijom):

(p \/ q) /\ ~~(~ p \/ s)~~ /\ (~s \/ ~q) /\ (s \/ p), PVal=[**~p**]

Jediničnom propagacijom dobijamo PVal=[**~p**, q]:

~~(p \/ q)~~ /\ ~~(~ p \/ s)~~ /\ (~s \/ ~q) /\ (s \/ p)

Novom jediničnom propagacijom dobijamo PVal=[**~p**, q, ~s] i dolazimo do konfliktne klauze *(s \/ p).* Sigurno je da treba da se vratimo unatrag, postavlja se samo pitanje odakle treba nastaviti pretragu. Treba primetiti da su literali *q* i *~s* dodati u parcijalnu valuaciju zbog **~p**. Pretraga gde mi iz parcijalne valuacije PVal=[**~p**, q, ~s] brišemo *~s* i dodajemo literal *s* nema smisla jer je klauza *(~s \/ ~q)* bila jedinična, ako izvršimo ovu zamenu sigurno postaje konfliktna. Ono što se zapravo dešava je to da su *q* i *~s* literali uslovljeni odlukom **~p**. Prilikom nasumičnog izbora literala *l* sve naredne jedinične propagacije do novog nasumičnog izbora literala su uslovljene literalom *l*. Dakle, jedino ima smisla vratiti se i promeniti **~p** u *p*. Dodatno, treba naglasiti da sada *p* nije nasumičan izbor - ako postoji, parcijalna valuacija koja je model za formulu mora da sadrži *p* zato što ne postoji nijedna parcijalna valuacija koja je model takva da sadrži *~p*.

Da sumiramo, kada naiđemo na konfliktnu klauzu bezbedno se možemo u pretrazi vratiti do poslednjeg nasumično izabranog literala i nastaviti pretragu odatle dodajući u parcijalnu valuaciju njemu suprotan literal.

Ovako dolazimo do iterativne implementacije koja koristi stek da na njemu čuva redosled literala dodatih u parcijalnu valuaciju. U beskonačnoj petlji proveravamo:

1. Ako smo naišli na konfliktnu klauzu
    a. Skidamo sa steka sve literale do poslednjeg nasumično izabranog literala *l* (poslednje izabranog hronološki, to je prvi takav kada stek gledamo odozgo na dole)
    b. Ako nema takvog literala (ekvivalentno ako je stek prazan jer smo ga ispraznili tražeći *l*)
        i. Vraćamo UNSAT
    c. Inače ubacujemo u parcijalnu valuaciju literal *~l* tj. suprotan literal od *l*
2. Inače, ako postoji jedinična klauza uradi propagaciju
3. Inače, ako postoji nedefinisani literal *l* dodaj ga u parcijalnu valuaciju
    a. Inače, ako takav ne postoji vrati SAT

Potrebno je malo prodiskutovati slučajeve kada algoritam vraća UNSAT odnosno SAT. Ako nema nijednog literala koji je slučajno odabran to znači da smo iscrpeli sve opcije i da nismo došli do modela za formulu, dakle ona je nezadovoljiva. S druge strane ako ne možemo da uradimo jediničnu propagaciju niti izaberemo nasumičan literal to znači da je parcijalna valuacija potpuna. Kada imamo potpunu parcijalnu valuaciju i nemamo konfliktnu klauzu to znači da imamo model za našu formulu.

DPLL algoritam se u literaturi često formuliše sledećim pravilima (*M* je parcijalna valuacija, *F* je formula, *C* klauza, a *l* literal):

- Jedinična propagacija (eng. unit propagation) - M || F, C \/ *l* → M*l* || F, C \/ *l*, kada važi M |= ~C i *l* je nedefinisan u M
- Čist literal (eng. pure literal) - M || F → M*l* || F, kada važi da se *l* pojavljuje u F i da se *~l* ne pojavljuje u F i *l* je nedefinisan u M

- Nasumičan izbor (eng. decide) - M || F → M/^{d} || F, kada važi se ~*l* ili *l* (ili oba) pojavljuje u F i da je *l* nedefinisan u M (*l* obeležavamo sa ^{d} u M da bi označili da je nasumično odabran)
- Neuspeh (eng. fail) - M || F, C → FailState, kada je M |= ~C i M ne sadrži nasumično odabran literal
- Povratak (eng. backtrack) - M/^{d}N || F, C → M~l || F, C, kada važi M/^{d}N |= ~C i N ne sadrži nasumično odabrane literale

Kada se DPLL opisuje formalno, navedena pravila se primenjuju tako da formiraju binarnu relaciju koja opisuje prelaske između stanja i na kraju konstruiše se sistem prelazaka. S obzirom na to da nema preke potrebe za formalizmima u cilju razumevanja samog algoritma, detaljna formulacija je izostavljena. Ipak čitalac bi trebalo da prepozna navedena pravila i njihove primene u opisu DPLL algoritma po koracima. Takođe, čitalac bi trebalo da nema nikakav problem da razume notaciju kojojm su pravila zapisana jer će ovakva notacija biti korišćena i u nastavku za neke druge teme.
Implementacija DPLL algoritma u programskom jeziku C++ je data u sekciji Implementacija.

Napredna poboljšanja

U jednom od prethodnih pasusa pomenuta je optimizaciona tehnika - šema posmatranja dva literala. Ideja je jako jednostavna, za svaku klauzu specijalno pamtimo 2 literala. Ako su oni nedefinisani važi da klauza ne može biti konfliktna niti može biti jedinična. Ako je jedan od njih tačan, klauza sigurno nije konfliktna. Kada se desi da jedan od dva nedefinisana literala postane netačan, tada je potrebna dodatna obrada, potrebno je naći naredni nedefinisani literal i prebaciti klauzu u njegovu listu klauza koje taj literal određuje. Dodatno, iz liste literala koji je postao netačan treba eliminisati tekuću klauzu. Ovde je opisana ideja algoritma, u zavisnosti od povratka u pretrazi potrebno je precizno definisati invarijante da se ne bi desio propust sa nekom klauzom ili literalom.
Kada se vraćamo u pretrazi do poslednjeg nasumično odabranog literala, mi preskačemo veliki broj permutacija vrednosti atoma za koje možemo da tvrdimo da nisu model i samim tim pretražujemo značajno manji prostor potencijalnih rešenja. Vraćanje u pretrazi do poslednjeg izabranog literala je prilično razuman korak, postavlja se pitanje da li možda možemo da se vratimo još više unazad?
Odgovor je da je nekada to moguće upotrebnom tehnika kao što su *povratni skok*, *objašnjavanje konflikata* i *učenje klauza*. Kada jednom dođe do konflikta mi nemamo garanciju da je do konflikta došlo zbog poslednjeg nivoa odlučivanja. Recimo da imamo ovakav sadržaj parcijalne valuacije (literali su dati u redosledu postavljanja na stek - *v* je poslednji dodat na stek) PVal=[**p**, q, r, **s**, **t**, u, v] i neka je konfliktna klauza *(~q \/ ~t \/ ~v)*. Naša pretraga bi mogla dva puta da udari u ovaj konflikt, jednom za PVal=[**p**, q, r, **s**, **t**, u, v] i drugi put za PVal=[**p**, q, r, **~s**, **t**, u, v]. Primetimo da promenljiva *s* uopšte ne učestvuje u konfliktu zbog čega su provere za literal *s* i za literal ~*s* irelevatne za konflikt koji nam se dešava. Izbegavanje ovih i sličnih situacija se rešava već pomenutim objašnjavanja konflikata, povratnim skokom i učenjem klauza. Postupak je sledeći, primenimo iskaznu rezoluciju nad skupom klauza *C* koji se formira od konfliktne klauze i svih onih klauza koje su propagirale literale u konfliktnoj klauzi. Rezultujuca klauza c_r je sigurno logička posledica postojećeg skupa klauza te je možemo bezbedno dodati skup postojećih klauza celokupne formule. U pretrazi, tj. na steku, se vraćamo do nivoa *d* takvog da svaki literal *l* iz c_r ima suprotan literal u parcijalnoj valuaciji osim jednog literala koji je nedefinisan. Možda nije odmah očigledno, ali kada nastavimo pretragu odatle prvi sledeći korak će biti propagacija nedefinisanog literala klauze c_r koju smo dodali u formulu. Učenjem ove klauze mi izbegavamo da završimo dva puta u istom konfliktu nakon potencijalno velikog broja koraka pretrage.

Dodatna poboljšanja uključuju heuristike za zaboravljanje naučenih klauza. Posle nekog vremena pretraga bi mogla da nauči previše klauza što bi dovelo do memorijskih problema i usporavanja pretrage. Osim zaboravljanja klauza korisna tehnika je i ponovno pokretanje pretrage. Intuicija je ta da kada smo naučili većinu problematičnih klauza, pretragom od početka ćemo brže i direktnije stići do cilja.
Jedina stvar kojoj do sada nismo posvetili puno pažnje i koju nismo optimizovali je nasumičan odabir literala koji ćemo dodati u parcijalnu valuaciju. Ispostavlja se da su neki literali podesniji za izabrati od drugih. Pre nego što izložimo heuristiku za odabir literala, prođimo kroz jedan hipotetički primer. Recimo da se nalazimo u nekom stanju pretrage S_i i da u narednih k koraka naučimo klauze $(\sim p_1 \vee p_2 \vee \sim p_3)$, $(p_2 \vee \sim p_{17} \vee \sim p_5)$, $(\sim p_4 \vee p_6 \vee p_2)$. Ako se nalazimo u stanju pretrage S_{i+k} i trebamo da odaberemo literal koji ćemo propagirati, da li možda imamo neki dobar izbor? Ideja je sledeća, literali koji učestvuju u naučenim klauzama su problematični - hajde da se pozabavimo njima da ne bismo kasnije dublje u pretrazi opet naišli na iste problematične literale (intuitivno smanjujemo prostor pretrage). U sve tri klauze iz primera se javlja literal p_2, on nam je *verovatno* najbolji izbor. Kada kažemo verovatno, to se odnosi na činjenicu da ne možemo sa sigurnošću znati da li je taj literal najbolji izbor, ali trebao bi da bude jako dobar izbor. Heuristika koja koristi ovu ideju se naziva ***V**ariable **S**tate **I**ndependent **D**ecaying **S**um* (u nastavku VSIDS). Za sve literale se čuva brojač inicijalizovan na nula. Kada se nauči nova klauza, svim literalima te klauze se uveća brojač. Vrednosti brojača se dele nekom konstantom nakon određenog broja iteracija. Ideja kod ovog skaliranja je da biramo literal koji je problematičan za trenutni prostor pretrage, a ne za prostor u kome smo bili pre mnogo iteracija.
Treba naglasiti da su u ovom tekstu date samo ideje i opisi visokog nivoa ovih naprednih tehnika, detalji potrebni za implementaciju su van obima časova vežbi. Detaljan i formalan pregled naprednih tehnika je dat u radu profesora Filipa Marića http://poincare.matf.bg.ac.rs/~filip//phd/sat-tutorial.pdf .

Implementacija

DIMACS format

Što se implementacije tiče, ulaz je formula u KNF i to u ispravnom DIMACS cnf formatu. DIMACS je tekstualni format. Linije sa komentarima počinju karakterom *c*. Prva linija ulaza koja nam je bitna je oblika:
p cnf broj_promenljivih broj_klauza
Nakon ove linije slede klauze koje su zadate svaka u svom redu. Između bilo koja dva reda mogu se naći prazne linije, odnosno linije sa komentarima. Klauza sadrži literale, u slučaju da je literal atom on se kodira pozitivnim brojem, u slučaju da je negacija atoma negacijom tog broja. Svaka linija koja sadrži klauzu se završava nulom. Primer ulaza:

```
c ovo je komentar
p cnf 3 4
-1 2 3 0
-2 3 0
-3 1 0
-2 0
```

odgovara formuli: $(\sim p_1 \vee p_2 \vee p_3) \wedge (\sim p_2 \vee p_3) \wedge (\sim p_3 \vee p_1) \wedge \sim p_2$.

Zaglavlja

partial_valuation.h

```
#ifndef PARTIALVALUATION_H
#define PARTIALVALUATION_H

#include <vector>
#include <cstdint>
#include <iostream>

/**
* S obzirom na to da cemo za literale koristiti oznacene celobrojne vrednosti 0 je
specijalna vrednost.
*/
#define NullLiteral (0)

/**
* @brief The Tribool enum - koristimo da kodiramo 3 vrednosti za promenljivu u
parcijalnoj valuaciji.
*
* @details Za razliku od C-a u C++-u je moguce kontrolisati tip enumeracije. Konkretno
nama
* trebaju 3 vrednosti zbog cega nam je 1 bajt i vise nego dovoljan.
*/
enum class Tribool: int8_t
{
 False,
 True,
 Undefined
};

/**
* Za literal cemo koristiti int vrednosti, pri cemu je najmanji indeks literala 1 tj.
p1.
* Ovo je zgodno iz 3 razloga, prvi je vezan za DIMACS format gde se promenljive
predstavljaju
* kao oznacene celobrojne vrednosti, drugi se odnosi na polaritet literala, odnosno na
to
* da li je literal na primer p1 ili ~p1 (1 koristimo za p1, a -1 za ~p1), a treci na
postojanje
* specijalne vrednosti koja ce igrati ulogu "rampe" u DPLL algoritmu (to je broj 0).
*/
using Literal = int;
using Clause = std::vector<Literal>;
using CNFFormula = std::vector<Clause>;

/**
* Deklaracija klase i operatora za ispis u stream.
*/
```

```
class PartialValuation;
std::ostream& operator<<(std::ostream &out, const PartialValuation &pval);

/**
* @brief The PartialValuation class - koristi se da predstavi parcijalnu valuaciju u
kojoj promenljive
* mogu biti tacne, netacne ili nedefinisane.
*/
class PartialValuation
{
public:
 PartialValuation(unsigned nVars = 0);
  /**
  * @brief push - na osnovu literala postavlja vrednost promenljive u parcijalnoj
valuaciji.
  * @param l - literal nas osnovu koga se postavlja vrednost
  * @param decide - ukoliko je 'decide' flag true oznacava se da je u pitanju decide
literal
  */
 void push(Literal l, bool decide = false);
  /**
  * @brief backtrack - skida literale sa steka sve do prvog decide literala na koji
naidje
  * @return poslednji decide literal ili NullLiteral ukoliko takvog nema
  */
 Literal backtrack();
  /**
  * @brief isClauseFalse - proverava da li je klauza netacna u tekucoj parcijalnoj
valuaciji.
  *
  * @details Klauza je netacna u tekucoj parcijalnoj valuaciji ako za svaki literal
klauze
  * vazi da je u parcijalnoj valuaciji njemu suprotan literal.
  * @param c - klauza koja se proverava
  * @return true ako je klauza netacna, false inace
  */
 bool isClauseFalse(const Clause &c) const;
  /**
  * @brief isClauseUnit - proverava da li je klauza jedinicna zbog propagacije
jedinicnog literala.
  *
  * @details Klauza je jedinicna ako za svaki literal klauze osim jednog, parcijalna
valuacija
  * sadrzi njemu suprotan literal. Ovaj jedan je nedefinisan.
  * @param c - klauza koja se proverava
  * @return literal koji je jedini nedefinisan u tekucoj klauzi
  */
 Literal isClauseUnit(const Clause &c) const;
  /**
  * @brief firstUndefined - trazi prvi nedefinisani literal u valuaciji
  *
```

```
  * @details Sluzi za izbor literala za decide pravilo. Ovde bi mogla da se ubaci neka
heuristika
  * za biranje najpodesnijeg literala, trenutno se uzima od svih nedefinisanih onaj sa
najmanjim
  * indeksom.
  * @return nedefinisani literal sa najmanjim indeksom
  */
 Literal firstUndefined() const;

 /**
  * @brief reset - postavlja parcijalnu valuaciju u pocetnu poziciju
  *
  * @details Sve promenljive se postavljaju na Tribool::Undefined, a stek se prazni.
  * @param nVars - broj promenljivih
  */
 void reset(unsigned nVars);
  friend std::ostream& operator<<(std::ostream &out, const PartialValuation &pval);
private:
 /**
  * @brief m_values - vrednost promenljivih u valuaciji
  */
 std::vector<Tribool> m_values;
 /**
  * @brief m_stack - stek na kome cuvamo istoriju postavljanja vrednosti promenljivih
zbog vracanja unazad
  */
 std::vector<Literal> m_stack;
};

#endif // PARTIALVALUATION_H
```

solver.h

```
#ifndef SOLVER_H
#define SOLVER_H

#include "partial_valuation.h"

#include <iostream>
#include <experimental/optional>

using OptionalPartialValuation = std::experimental::optional<PartialValuation>;

class Solver
{
public:
 /**
  * @brief Solver - konstruktor od CNF formule
  * @param formula - CNF za koji proveravamo zadovoljivost
  */
 Solver(const CNFFormula &formula);
  /**
```

```
  * @brief Solver - konstruktor od C++ stream-a iz koga se cita CNF u DIMACS formatu
  * @param dimacsStream - ulazni stream
  */
 Solver(std::istream &dimacsStream);
  /**
  * @brief solve - DPLL algoritam za proveru zadovoljivosti
  * @return parcijalnu valuaciju ili nista
  */
 OptionalPartialValuation solve();
 private:
 /**
  * @brief hasConflict - provera da li je neka klauza u konfliktu sa tekucom valuacijom
  * @return true ako je neka klauza u konfliktu, false inace
  */
 bool hasConflict() const;
  /**
  * @brief hasUnitClause
  * @return
  */
 Literal hasUnitClause() const;
 private:
 CNFFormula m_formula;
 PartialValuation m_valuation;
};

#endif // SOLVER_H
```

Izvorni kodovi

partial_valuation.cpp

```
#include "partial_valuation.h"

#include <algorithm>
#include <cstdlib>

PartialValuation::PartialValuation(unsigned nVars)
 : m_values(nVars+1, Tribool::Undefined),
   m_stack()
{
 m_stack.reserve(nVars); /* rezervisemo memoriju da izbegnemo ceste realokacije */
}

void PartialValuation::push(Literal l, bool decide)
{
 /* Promenljivu od literala dobijamo sa std::abs, a za polaritet proveravamo znak */
 m_values[std::abs(l)] = l > 0 ? Tribool::True : Tribool::False;
 if (decide)
 {
   /* Postavljamo rampu pre literala da bismo znali da je decide literal u pitanju */
   m_stack.push_back(NullLiteral);
```

```
 }
 m_stack.push_back(l);
}

Literal PartialValuation::backtrack()
{
 /* Proveravamo da nije prazan stek */
 if (m_stack.empty())
 {
   return NullLiteral;
 }
  Literal lastDecide = NullLiteral, last = NullLiteral;
 do {
   /* Dohvatamo poslednje postavljeni literal i skidamo ga sa steka */
   last = m_stack.back();
   m_stack.pop_back();
   m_values[std::abs(last)] = Tribool::Undefined;

   /* Ako je on NullLiteral, tj. rampa vracamo lastDecide literal */
   if (NullLiteral == last)
   {
     break;
   }
   lastDecide = last;
 } while (m_stack.size());
  return last == NullLiteral ? lastDecide : NullLiteral;
}

bool PartialValuation::isClauseFalse(const Clause &c) const
{
 /* Za svaki literal klauze proveravamo da li se njegov suprotni nalazi u parc. val. */
 for (Literal l : c)
 {
   Tribool variableInClause = l > 0 ? Tribool::True : Tribool::False;
   Tribool variableInValuation = m_values[std::abs(l)];
   if (variableInClause == variableInValuation || variableInValuation ==
Tribool::Undefined)
   {
     return false;
   }
 }
 return true;
}

Literal PartialValuation::isClauseUnit(const Clause &c) const
{
 /* Definisemo promenljive za broj nedefinisanih literala i poslednji nedefinisani
literal */
 Literal undefinedLit = NullLiteral;
 int cntUndefined = 0;
  /* Za svaki literal proveravamo da li je nedefinisan */
```

```
 for (Literal l : c)
 {
   Tribool valueInClause = l > 0 ? Tribool::True : Tribool::False;
   Tribool valueInValuation = m_values[std::abs(l)];
   if (valueInClause != valueInValuation)
   {
     if (valueInValuation == Tribool::Undefined)
     {
       ++cntUndefined;
       undefinedLit = l;

       /* Ako naidjemo na jos jedan nedefinisan literal - klauza nije jedinicna */
       if (cntUndefined > 1)
       {
         break;
       }
     }
   }
   else
   {
     return NullLiteral;
   }
 }
 return cntUndefined == 1 ? undefinedLit : NullLiteral;
}

Literal PartialValuation::firstUndefined() const
{
 auto it = std::find(m_values.cbegin()+1, m_values.cend(), Tribool::Undefined);
 return it != m_values.cend() ? it-m_values.cbegin() : NullLiteral;
}

void PartialValuation::reset(unsigned nVars)
{
 m_values.resize(nVars+1);
 std::fill(m_values.begin(), m_values.end(), Tribool::Undefined);
  m_stack.clear();
 m_stack.reserve(nVars);
}


std::ostream &operator<<(std::ostream &out, const PartialValuation &pval)
{
 out << "[ ";
 for (std::size_t i = 1; i < pval.m_values.size(); ++i)
 {
   if (pval.m_values[i] == Tribool::True)
   {
     out << 'p' << i << ' ';
   }
   else if (pval.m_values[i] == Tribool::False)
```

```
    {
      out << "~p" << i << ' ';
    }
    else if (pval.m_values[i] == Tribool::Undefined)
    {
      out << 'u' << i << ' ';
    }
    else
    {
      throw std::logic_error{"Nepoznata vrednost dodeljena promenljivoj (nije ni True, ni
False, ni Undefined)"};
    }
  }
  return out << " ]";
}
```

solver.cpp

```
#include "solver.h"

#include <string>
#include <sstream>
#include <stdexcept>
#include <iterator>

Solver::Solver(const CNFFormula &formula)
  : m_formula(formula), m_valuation(m_formula.size())
{
}

Solver::Solver(std::istream &dimacsStream)
{
  /* Citamo uvodne komentare, preskacemo prazne linije */
  std::string line;
  std::size_t firstNonSpaceIdx;
  while (std::getline(dimacsStream, line))
  {
    firstNonSpaceIdx = line.find_first_not_of(" \t\r\n");
    if (firstNonSpaceIdx != std::string::npos && line[firstNonSpaceIdx] != 'c')
    {
      break;
    }
  }
   /* Proveravamo da smo procitali liniju 'p cnf brPromenljivih brKlauza' */
  if (line[firstNonSpaceIdx] != 'p')
  {
    throw std::runtime_error{"Pogresan format ulaza iz DIMACS stream-a"};
  }
  std::istringstream parser{line.substr(firstNonSpaceIdx+1, std::string::npos)};
  std::string tmp;
  if (!(parser >> tmp) || tmp != "cnf")
  {
```

```
    throw std::runtime_error{"Pogresan format ulaza iz DIMACS stream-a"};
  }
  unsigned varCnt, claCnt;
  if (!(parser >> varCnt >> claCnt))
  {
    throw std::runtime_error{"Pogresan format ulaza iz DIMACS stream-a"};
  }
   /* Citamo klauze linije po liniju preskacuci komentare i prazne linije */
  m_valuation.reset(varCnt);
  m_formula.resize(claCnt);
  int clauseIdx = 0;
  while (std::getline(dimacsStream, line))
  {
    firstNonSpaceIdx = line.find_first_not_of(" \t\r\n");
    if (firstNonSpaceIdx != std::string::npos && line[firstNonSpaceIdx] != 'c')
    {
      parser.clear();
      parser.str(line);
      std::copy(std::istream_iterator<int>{parser}, {},
std::back_inserter(m_formula[clauseIdx]));
      m_formula[clauseIdx++].pop_back(); /* izbacujemo nulu sa kraja linije */
    }
  }
}

OptionalPartialValuation Solver::solve()
{
  while (true)
  {
    /* Proverimo da li ima bar 1 konflikt */
    Literal l;
    if (hasConflict())
    {
      /* Radimo backtracking i dobijamo literal koji smo nekad ranije postavili */
      Literal decidedLiteral = m_valuation.backtrack();
      if (NullLiteral == decidedLiteral)
      {
        /* Ne mozemo vise da radimo backtrack, iscrpeli smo sve valuacije */
        return {};
      }

      /* Postavljamo suprotnu vrednost literala i nastavljamo */
      m_valuation.push(-decidedLiteral);
    }
    else if ((l = hasUnitClause()))
    {
      /* Propagacija jedinicne klauze */
      m_valuation.push(l);
    }
    else
    {
```

```
      /* Decide pravilo */
      if ((l = m_valuation.firstUndefined()))
      {
        m_valuation.push(l, true);
      }
      else
      {
        /* Ne mozemo da primenimo decide jer imamo punu valuaciju */
        return m_valuation;
      }
    }
  }
}

bool Solver::hasConflict() const
{
  for (const Clause &c : m_formula)
  {
    if (m_valuation.isClauseFalse(c))
    {
      return true;
    }
  }
  return false;
}

Literal Solver::hasUnitClause() const
{
  Literal l;
  for (const Clause &c : m_formula)
  {
    if ((l = m_valuation.isClauseUnit(c)))
    {
      return l;
    }
  }
   return 0;
}
```

Čas 4 - MiniSAT i kodiranje raznih problema

Preduslovi: *čas 1 (samo poznavanje formula iskazne logike); čas 2 poznavanje transformacija u KNF; poznavanje logičkih kola (predmet Uvod u organizaciju i arhitekturu računara 2);*

Iako je spektar primene SAT rešavača zaista veliki, kodiranje problema tako da se od instance dobije formula iskazne logike nije algoritamski definisan postupak u opštem slučaju. Za ispravno kodiranje problema u SAT je potrebno razviti određeno iskustvo, kojim se dobija intuicija za prepoznavanje kodiranje za novi problem na osnovu slične ili iste šeme kodiranja koju smo već

videli na nekom drugom problemu. U nastavku će biti prikazana kodiranja za nekoliko vrsta jednostavnih problema u cilju razvijanja ove veštine.
Alat koji ćemo pokretati u cilju rešavanja SAT problema je MiniSAT. Ovo je slobodno dostupan rešavač koji se aktivno održava i u kome je implementiran veliki broj naprednih tehnika za rešavanje SAT problema (potsetiti se Napredna poboljšanja). Na Linux distribucijama najčešće postoji odgovarajući paket koji je moguće instalirati putem komandne linije:

- Ubuntu zasnovane distribucije: *sudo apt-get install minisat*

I za druge distribucije paket će se najverovatnije zvati *minisat*. U *man* stranama se mogu dobiti detaljne informacije o korišćenju *minisat* alata. Osnovna upotreba je:

minisat datoteka.cnf model.out

Pri čemu *datoteka.cnf* sadrži formulu u DIMACS cnf formatu, dok će se u fajlu *model.out* naći model za formulu ukoliko takav postoji. Informacije o zadovoljivosti formule, kao i statistike o izvršavanju *minisat* će ispisati na standardni izlaz.
Nekada problemi zahtevaju proveru jedinstvenosti rešenja. U tim slučajevima postoji prost način da proverimo da li je rešenje jedinstveno. Iz fajla *model.out* uzmemo valuaciju koja je model za tekuću formulu i dodamo klauzu formiranu od ove valuacije tako što je uzet suprotan literal za svaki literal valuacije. Treba biti pažljiv i uvećati broj klauza za jedan u DIMACS datoteci. Pokrenemo *minisat* za izmenjenu formulu i ako pretraga nađe novi model rešenje nije jedinstveno, dok ako *minisat* vrati UNSAT znači da je prvo pronađeno rešenje ujedno i jedino.

Kodiranje logičkih kola

Primer 1: Neka je dat dvocifreni binarni brojač:

- Konstruisati formulu iskazne logike kojom se opisuje prelaz između susednih stanja p_iq_i - > $p_{i+1}q_{i+i}$
- Dokazati da se nakon 4 prelaza dobije početno stanje

Rešenje:

Ispišimo stanja dvocifrenog brojača kako bismo uočili neke pravilnosti:
p_iq_i $p_{i+1}q_{i+1}$
00 -> 01
01 -> 10
10 -> 11
11 -> 00
Bit najmanje težine se uvek menja odakle zaključujemo:
$q_i <=> \sim q_{i+1}$
Bit najveće težine tj. p_{i+1} je 1 akko su bitovi p_i i q_i različiti inače je 0:
$(q_i \wedge \sim p_i) \vee (\sim q_i \wedge p_i) <=> p_{i+1}$
Dalje je potrebno pokazati da $p_1 <=> p_5$ i $q_1 <=> q_5$ ako važe gore navedene veze. To faktički znači da su ove dve ekvivalencije posledice gornjih veza. Da je formula posledica skupa formula izražavamo implikacijom. Kada razbijemo gornje veze u KNF dobijamo:

$(q_i \vee q_{i+1}) \wedge (\sim q_i \vee \sim q_{i+1}) \wedge$
$(\sim p_{i+1} \vee \sim q_i \vee \sim p_i) \wedge$
$(p_{i+1} \vee q_i \vee \sim p_i) \wedge$
$(\sim p_{i+1} \vee q_i \vee p_i) \wedge$
$(p_{i+1} \vee \sim q_i \vee p_i)$

Pokazaćemo da je naša implikacija tautologija tako što ćemo uzeti njenu negaciju i dokazati da je nezadovoljiva korišćenjem *minisat* rešavača. S obzirom na to da se implikacija rastavlja kao *a => b -> ~a \/ b*, a da mi uzimamo njenu negaciju dobijamo *~(a => b) -> a /\ ~b*. Dakle KNF konstruišemo tako što od veza između p_i, p_{i+1} i q_i, q_{i+1} formiramo KNF i nadovežemo KNF od negirane desne strane implikacije. Raspisano za i=1,5 to izgleda ovako:

$(q_1 \vee q_2) \wedge$
$(\neg q_1 \vee \neg q_2) \wedge$
$(\neg p_2 \vee \neg q_1 \vee \neg p_1) \wedge$
$(p_2 \vee q_1 \vee \neg p_1) \wedge$
$(\neg p_2 \vee q_1 \vee p_1) \wedge$
$(p_2 \vee \neg q_1 \vee p_1)$
$(q_2 \vee q_3) \wedge$
$(\neg q_2 \vee \neg q_3) \wedge$
$(\neg p_3 \vee \neg q_2 \vee \neg p_2) \wedge$
$(p_3 \vee q_2 \vee \neg p_2) \wedge$
$(\neg p_3 \vee q_2 \vee p_2) \wedge$
$(p_3 \vee \neg q_2 \vee p_2)$
$(q_3 \vee q_4) \wedge$
$(\neg q_3 \vee \neg q_4) \wedge$
$(\neg p_4 \vee \neg q_3 \vee \neg p_3) \wedge$
$(p_4 \vee q_3 \vee \neg p_3) \wedge$
$(\neg p_4 \vee q_3 \vee p_3) \wedge$
$(p_4 \vee \neg q_3 \vee p_3)$
$(q_4 \vee q_5) \wedge$
$(\neg q_4 \vee \neg q_5) \wedge$
$(\neg p_5 \vee \neg q_4 \vee \neg p_4) \wedge$
$(p_5 \vee q_4 \vee \neg p_4) \wedge$
$(\neg p_5 \vee q_4 \vee p_4) \wedge$
$(p_5 \vee \neg q_4 \vee p_4) \wedge$
$(\neg p_1 \vee \neg p_5 \vee \neg q_1 \vee \neg q_5) \wedge$
$(p_1 \vee p_5 \vee \neg q_1 \vee \neg q_5) \wedge$
$(\neg p_1 \vee \neg p_5 \vee q_1 \vee q_5) \wedge$
$(p_1 \vee p_5 \vee q_1 \vee q_5)$

Iskošene klauze predstavljaju negiranu desnu stranu implikacije. Ukupno imamo 10 promenljivih i 28 klauza.

Primer 2: Neka je dat četvorobitni broj *x* u potpunom komplementu. Pokazati da kada se dva puta primeni operacija komplementiranja nad *x* da se dobije početni broj - *(x')' = x*. Operacija komplementiranja se radi tako što se svi bitovi izvrnu i na dobijeni broj se doda 1.

Rešenje:

Ponovo započnimo primerima komplementiranja:

0000 -> 1111 -> 0000
0001 -> 1110 -> 1111
0010 -> 1101 -> 1110
0011 -> 1100 -> 1101
0100 -> 1011 -> 1100
0101 -> 1010 -> 1011

0110 -> 1001 -> 1010
0111 -> 1000 -> 1001
1000 -> 0111 -> 1000
...
Primetimo da se bit najmanje težine ne menja. Bitove početnog broja ćemo obeležavati sa $x_3x_2x_1x_0$, a bitove nakon prvog potpunog komplementa sa $y_3y_2y_1y_0$ i na kraju bitove nakon drugog potpunog komplementa sa $z_3z_2z_1z_0$. U toj notaciji možemo napisati identitet:
$x_0 <=> y_0$
koji se rastavlja u:
$(\sim x_0 \setminus/ y_0) /\setminus$
$(x_0 \setminus/ \sim y_0)$
Što se tiče bita y_1 on zavisi od bitova na poziciji x_0 i poziciji x_1, praktično $y_1 = x_0 \hat{} x_1$ gde je ^ operacija ekskluzivne disjunkcije što prevodimo u:
$y_1 <=> (\sim x_0 /\setminus x_1) \setminus/ (x_0 /\setminus \sim x_1)$ ili u ekvivalentan identitet
$y_1 <=> \underline{(\sim x_0 \setminus/ \sim x_1) /\setminus (x_0 \setminus/ x_1)}$
Kada imamo ekvivalenciju *a* <=> *b* mi je možemo rastaviti na:
(~a \/ b) /\ (a \/ ~b)
Ulogu *a* igra y_1, ulogu *b* podvučena desna strana, a ulogu *~b* desna strana iz prve ekvivalencije:
$(\sim y_1 \setminus/ \{(\sim x_0 \setminus/ \sim x_1) /\setminus (x_0 \setminus/ x_1)\}) /\setminus (y_1 \setminus/ \sim\{(\sim x_0 /\setminus x_1) \setminus/ (x_0 /\setminus \sim x_1)\})$
što daje:
$\{(\sim y_1 \setminus/ \sim x_0 \setminus/ \sim x_1) /\setminus (\sim y_1 \setminus/ x_0 \setminus/ x_1)\} /\setminus$
$\{(y_1 \setminus/ x_0 \setminus/ \sim x_1) /\setminus (y_1 \setminus/ \sim x_0 \setminus/ x_1)\}$
i na kraju:
$(\sim y_1 \setminus/ \sim x_0 \setminus/ \sim x_1) /\setminus$
$(\sim y_1 \setminus/ x_0 \setminus/ x_1) /\setminus$
$(y_1 \setminus/ x_0 \setminus/ \sim x_1) /\setminus$
$(y_1 \setminus/ \sim x_0 \setminus/ x_1)$
E sad za y_2 treba posmatrati 3 bita x_0, x_1 i x_2. Postoji način da izbegnemo mozganje kako formirati izraz. Fokusirajmo se na logičku tablicu koja se dobija kada posmatramo vrednosti y_2 u zavisnosti od x_0, x_1 i x_2:
$x_2x_1x_0 \mid y_2$
000 -> 0 --> $\underline{(x_0 \setminus/ x_1 \setminus/ x_2)}$
001 -> 1 --> $(x_0 /\setminus \sim x_1 /\setminus \sim x_2)$
010 -> 1 --> $(\sim x_0 /\setminus x_1 /\setminus \sim x_2)$
011 -> 1 --> $(x_0 /\setminus x_1 /\setminus \sim x_2)$
100 -> 1 --> $(\sim x_0 /\setminus \sim x_1 /\setminus x_2)$
101 -> 0 --> $\underline{(\sim x_0 \setminus/ x_1 \setminus/ \sim x_2)}$
110 -> 0 --> $\underline{(x_0 \setminus/ \sim x_1 \setminus/ \sim x_2)}$
111 -> 0 --> $\underline{(\sim x_0 \setminus/ \sim x_1 \setminus/ \sim x_2)}$
Mi odavde direktno možemo da izvučemo dve ekvivalencije:
$y_2 <=> (x_0 /\setminus \sim x_1 /\setminus \sim x_2) \setminus/ (\sim x_0 /\setminus x_1 /\setminus \sim x_2) \setminus/ (x_0 /\setminus x_1 /\setminus \sim x_2) \setminus/ (\sim x_0 /\setminus \sim x_1 /\setminus x_2)$
$y_2 <=> (x_0 \setminus/ x_1 \setminus/ x_2) /\setminus (\sim x_0 \setminus/ x_1 \setminus/ \sim x_2) /\setminus (x_0 \setminus/ \sim x_1 \setminus/ \sim x_2) /\setminus (\sim x_0 \setminus/ \sim x_1 \setminus/ \sim x_2)$
U pitanju su kanonska DNF odnosno kanonska KNF. Sad iskoristimo obe ekvivalencije na pravom mestu:
$(\sim y_2 \setminus/ \{(x_0 \setminus/ x_1 \setminus/ x_2) /\setminus (\sim x_0 \setminus/ x_1 \setminus/ \sim x_2) /\setminus (x_0 \setminus/ \sim x_1 \setminus/ \sim x_2) /\setminus (\sim x_0 \setminus/ \sim x_1 \setminus/ \sim x_2)\}) /\setminus$

$(y_2 \vee \sim\{(x_0 \wedge \sim x_1 \wedge \sim x_2) \vee (\sim x_0 \wedge x_1 \wedge \sim x_2) \vee (x_0 \wedge x_1 \wedge \sim x_2) \vee (\sim x_0 \wedge \sim x_1 \wedge x_2)\})$
što daje:
$(\sim y_2 \vee \{(x_0 \vee x_1 \vee x_2) \wedge (\sim x_0 \vee x_1 \vee \sim x_2) \wedge (x_0 \vee \sim x_1 \vee \sim x_2) \wedge (\sim x_0 \vee \sim x_1 \vee \sim x_2)\}) \wedge$
$(y_2 \vee \{(\sim x_0 \vee x_1 \vee x_2) \wedge (x_0 \vee \sim x_1 \vee x_2) \wedge (\sim x_0 \vee \sim x_1 \vee x_2) \wedge (x_0 \vee x_1 \vee \sim x_2)\})$
i na kraju dobijamo:
$(\sim y_2 \vee x_0 \vee x_1 \vee x_2) \wedge$
$(\sim y_2 \vee \sim x_0 \vee x_1 \vee \sim x_2) \wedge$
$(\sim y_2 \vee x_0 \vee \sim x_1 \vee \sim x_2) \wedge$
$(\sim y_2 \vee \sim x_0 \vee \sim x_1 \vee \sim x_2) \wedge$
$(y_2 \vee \sim x_0 \vee x_1 \vee x_2) \wedge$
$(y_2 \vee x_0 \vee \sim x_1 \vee x_2) \wedge$
$(y_2 \vee \sim x_0 \vee \sim x_1 \vee x_2) \wedge$
$(y_2 \vee x_0 \vee x_1 \vee \sim x_2)$
Zbog y3 ispišimo celokupnu tablicu:

$x_3x_2x_1x_0$	y_3
0000	-> 0
0001	-> 1
0010	-> 1
0011	-> 1
0100	-> 1
0101	-> 1
0110	-> 1
0111	-> 1
1000	-> 1
1001	-> 0
1010	-> 0
1011	-> 0
1100	-> 0
1101	-> 0
1110	-> 0
1111	-> 0

Na isti način kao za y2 možemo formirati 16 klauza:
$(\sim y_3 \vee x_3 \vee x_2 \vee x_1 \vee x_0) \wedge$
$(\sim y_3 \vee \sim x_3 \vee x_2 \vee x_1 \vee \sim x_0) \wedge$
$(\sim y_3 \vee \sim x_3 \vee x_2 \vee \sim x_1 \vee x_0) \wedge$
$(\sim y_3 \vee \sim x_3 \vee x_2 \vee \sim x_1 \vee \sim x_0) \wedge$
$(\sim y_3 \vee \sim x_3 \vee \sim x_2 \vee x_1 \vee x_0) \wedge$
$(\sim y_3 \vee \sim x_3 \vee \sim x_2 \vee x_1 \vee \sim x_0) \wedge$
$(\sim y_3 \vee \sim x_3 \vee \sim x_2 \vee \sim x_1 \vee x_0) \wedge$
$(\sim y_3 \vee \sim x_3 \vee \sim x_2 \vee \sim x_1 \vee \sim x_0) \wedge$
$(y_3 \vee x_3 \vee x_2 \vee x_1 \vee \sim x_0) \wedge$
$(y_3 \vee x_3 \vee x_2 \vee \sim x_1 \vee x_0) \wedge$
$(y_3 \vee x_3 \vee x_2 \vee \sim x_1 \vee \sim x_0) \wedge$
$(y_3 \vee x_3 \vee \sim x_2 \vee x_1 \vee x_0) \wedge$
$(y_3 \vee x_3 \vee \sim x_2 \vee x_1 \vee \sim x_0) \wedge$
$(y_3 \vee x_3 \vee \sim x_2 \vee \sim x_1 \vee x_0) \wedge$

$(y_3 \vee x_3 \vee \neg x_2 \vee \neg x_1 \vee \neg x_0) \wedge$
$(y_3 \vee \neg x_3 \vee x_2 \vee x_1 \vee x_0)$

S obzirom na to da je odnos y_i i z_i isti kao i odnos između x_i i y_i ove klauze se dupliraju sa odgovarajućim zamenama promenljivih i dobijamo:

$(\neg x_0 \vee y_0) \wedge$
$(x_0 \vee \neg y_0)$
$\wedge$
$(\neg y_1 \vee \neg x_0 \vee \neg x_1) \wedge$
$(\neg y_1 \vee x_0 \vee x_1) \wedge$
$(y_1 \vee x_0 \vee \neg x_1) \wedge$
$(y_1 \vee \neg x_0 \vee x_1)$
$\wedge$
$(\neg y_2 \vee x_0 \vee x_1 \vee x_2) \wedge$
$(\neg y_2 \vee \neg x_0 \vee x_1 \vee \neg x_2) \wedge$
$(\neg y_2 \vee x_0 \vee \neg x_1 \vee \neg x_2) \wedge$
$(\neg y_2 \vee \neg x_0 \vee \neg x_1 \vee \neg x_2) \wedge$
$(y_2 \vee \neg x_0 \vee x_1 \vee x_2) \wedge$
$(y_2 \vee x_0 \vee \neg x_1 \vee x_2) \wedge$
$(y_2 \vee \neg x_0 \vee \neg x_1 \vee x_2) \wedge$
$(y_2 \vee x_0 \vee x_1 \vee \neg x_2)$
$\wedge$
$(\neg y_3 \vee x_3 \vee x_2 \vee x_1 \vee x_0) \wedge$
$(\neg y_3 \vee \neg x_3 \vee x_2 \vee x_1 \vee \neg x_0) \wedge$
$(\neg y_3 \vee \neg x_3 \vee x_2 \vee \neg x_1 \vee x_0) \wedge$
$(\neg y_3 \vee \neg x_3 \vee x_2 \vee \neg x_1 \vee \neg x_0) \wedge$
$(\neg y_3 \vee \neg x_3 \vee \neg x_2 \vee x_1 \vee x_0) \wedge$
$(\neg y_3 \vee \neg x_3 \vee \neg x_2 \vee x_1 \vee \neg x_0) \wedge$
$(\neg y_3 \vee \neg x_3 \vee \neg x_2 \vee \neg x_1 \vee x_0) \wedge$
$(\neg y_3 \vee \neg x_3 \vee \neg x_2 \vee \neg x_1 \vee \neg x_0) \wedge$
$(y_3 \vee x_3 \vee x_2 \vee x_1 \vee \neg x_0) \wedge$
$(y_3 \vee x_3 \vee x_2 \vee \neg x_1 \vee x_0) \wedge$
$(y_3 \vee x_3 \vee x_2 \vee \neg x_1 \vee \neg x_0) \wedge$
$(y_3 \vee x_3 \vee \neg x_2 \vee x_1 \vee x_0) \wedge$
$(y_3 \vee x_3 \vee \neg x_2 \vee x_1 \vee \neg x_0) \wedge$
$(y_3 \vee x_3 \vee \neg x_2 \vee \neg x_1 \vee x_0) \wedge$
$(y_3 \vee x_3 \vee \neg x_2 \vee \neg x_1 \vee \neg x_0) \wedge$
$(y_3 \vee \neg x_3 \vee x_2 \vee x_1 \vee x_0)$
$\wedge$
$(\neg y_0 \vee z_0) \wedge$
$(y_0 \vee \neg z_0)$
$\wedge$
$(\neg z_1 \vee \neg y_0 \vee \neg y_1) \wedge$
$(\neg z_1 \vee y_0 \vee y_1) \wedge$
$(z_1 \vee y_0 \vee \neg y_1) \wedge$
$(z_1 \vee \neg y_0 \vee y_1)$
$\wedge$

$(\sim z_2 \vee y_0 \vee y_1 \vee y_2) \wedge$
$(\sim z_2 \vee \sim y_0 \vee y_1 \vee \sim y_2) \wedge$
$(\sim z_2 \vee y_0 \vee \sim y_1 \vee \sim y_2) \wedge$
$(\sim z_2 \vee \sim y_0 \vee \sim y_1 \vee \sim y_2) \wedge$
$(z_2 \vee \sim y_0 \vee y_1 \vee y_2) \wedge$
$(z_2 \vee y_0 \vee \sim y_1 \vee y_2) \wedge$
$(z_2 \vee \sim y_0 \vee \sim y_1 \vee y_2) \wedge$
$(z_2 \vee y_0 \vee y_1 \vee \sim y_2)$
$\wedge$
$(\sim z_3 \vee y_3 \vee y_2 \vee y_1 \vee y_0) \wedge$
$(\sim z_3 \vee \sim y_3 \vee y_2 \vee y_1 \vee \sim y_0) \wedge$
$(\sim z_3 \vee \sim y_3 \vee y_2 \vee \sim y_1 \vee y_0) \wedge$
$(\sim z_3 \vee \sim y_3 \vee y_2 \vee \sim y_1 \vee \sim y_0) \wedge$
$(\sim z_3 \vee \sim y_3 \vee \sim y_2 \vee y_1 \vee y_0) \wedge$
$(\sim z_3 \vee \sim y_3 \vee \sim y_2 \vee y_1 \vee \sim y_0) \wedge$
$(\sim z_3 \vee \sim y_3 \vee \sim y_2 \vee \sim y_1 \vee y_0) \wedge$
$(\sim z_3 \vee \sim y_3 \vee \sim y_2 \vee \sim y_1 \vee \sim y_0) \wedge$
$(z_3 \vee y_3 \vee y_2 \vee y_1 \vee \sim y_0) \wedge$
$(z_3 \vee y_3 \vee y_2 \vee \sim y_1 \vee y_0) \wedge$
$(z_3 \vee y_3 \vee y_2 \vee \sim y_1 \vee \sim y_0) \wedge$
$(z_3 \vee y_3 \vee \sim y_2 \vee y_1 \vee y_0) \wedge$
$(z_3 \vee y_3 \vee \sim y_2 \vee y_1 \vee \sim y_0) \wedge$
$(z_3 \vee y_3 \vee \sim y_2 \vee \sim y_1 \vee y_0) \wedge$
$(z_3 \vee y_3 \vee \sim y_2 \vee \sim y_1 \vee \sim y_0) \wedge$
$(z_3 \vee \sim y_3 \vee y_2 \vee y_1 \vee y_0)$

Ostaje nam još da razvijemo u KNF formulu ~Q to jest, da pokažemo da je ~(P => Q) -> P /\ ~Q nezadovoljiva pri čemu je P skup navedenih klauza, a ~Q je negacija konjunkcije sledećih ekvivalencija (ekvivalencijama pokazujemo da smo se vratili u početno stanje):

$\sim((z_0 <=> x_0) \wedge (z_1 <=> x_1) \wedge (z_2 <=> x_2) \wedge (z_3 <=> x_3))$

Ovo je bitan trenutak u rešavanju zadatka. Primena direktnih transformacija na ~Q u cilju dolaska do KNF će dovesti do kombinatorne eksplozije i ogromnog broja klauza. Zainteresovani čitalac može kopirati prilagođenu formulu ~Q uz izmeni simbola za veznike na neki od Veb sajtova za automatsko prevođenje u KNF da se uveri u to (jedan primer sajta je http://formal.cs.utah.edu:8080/pbl/PBL.php). U ovakvoj situaciji potrebno je primeniti Cajtinovu transformaciju, za ~Q:

$\sim(x_0 <=> z_0) \vee \sim(x_1 <=> z_1) \vee \sim(x_2 <=> z_2) \vee \sim(x_3 <=> z_3)$

uvodimo zamene:

$u_0 <=> \sim(x_0 <=> z_0)$
$u_1 <=> \sim(x_1 <=> z_1)$,
$u_2 <=> \sim(x_2 <=> z_2)$
$u_3 <=> \sim(x_3 <=> z_3)$

gde se zamena rastavlja na (primer je za u_0, x_0 i z_0):

$(\sim u_0 \vee x_0 \vee z_0) \wedge$
$(\sim u_0 \vee x_0 \vee \sim x_0) \wedge$
$(\sim u_0 \vee \sim z_0 \vee z_0) \wedge$
$(\sim u_0 \vee \sim z_0 \vee \sim x_0) \wedge$

$(\sim x_0 \vee z_0 \vee u_0) \wedge$
$(\sim z_0 \vee x_0 \vee u_0)$
pri čemu se podvučene klauze eliminišu jer sadrže literal i njemu suprotan literal i ostaju:
$(\sim u_0 \vee x_0 \vee z_0) \wedge$
$(\sim u_0 \vee \sim z_0 \vee \sim x_0) \wedge$
$(\sim x_0 \vee z_0 \vee u_0) \wedge$
$(\sim z_0 \vee x_0 \vee u_0)$
Zbog kompletnosti, prikažemo formulu ~Q nakon zamena:
$(u_0 \vee u_1 \vee u_2 \vee u_3) \wedge$
$u_0 \Leftrightarrow \sim(x_0 \Leftrightarrow z_0) \wedge$
$u_1 \Leftrightarrow \sim(x_1 \Leftrightarrow z_1) \wedge$
$u_2 \Leftrightarrow \sim(x_2 \Leftrightarrow z_2) \wedge$
$u_3 \Leftrightarrow \sim(x_3 \Leftrightarrow z_3)$
Sa raspisanim zamenama (kao za u_0, x_0 i z_0) dobijaju se sledeće klauze za ~Q:
$(u_0 \vee u_1 \vee u_2 \vee u_3) \wedge$
$(\sim u_0 \vee x_0 \vee z_0) \wedge$
$(\sim u_0 \vee \sim z_0 \vee \sim x_0) \wedge$
$(\sim x_0 \vee z_0 \vee u_0) \wedge$
$(\sim z_0 \vee x_0 \vee u_0) \wedge$
$(\sim u_1 \vee x_1 \vee z_1) \wedge$
$(\sim u_1 \vee \sim z_1 \vee \sim x_1) \wedge$
$(\sim x_1 \vee z_1 \vee u_1) \wedge$
$(\sim z_1 \vee x_1 \vee u_1) \wedge$
$(\sim u_2 \vee x_2 \vee z_2) \wedge$
$(\sim u_2 \vee \sim z_2 \vee \sim x_2) \wedge$
$(\sim x_2 \vee z_2 \vee u_2) \wedge$
$(\sim z_2 \vee x_2 \vee u_2) \wedge$
$(\sim u_3 \vee x_3 \vee z_3) \wedge$
$(\sim u_3 \vee \sim z_3 \vee \sim x_3) \wedge$
$(\sim x_3 \vee z_3 \vee u_3) \wedge$
$(\sim z_3 \vee x_3 \vee u_3)$
Konačno sve klauze su:
$(\sim x_0 \vee y_0) \wedge$
$(x_0 \vee \sim y_0)$
$\wedge$
$(\sim y_1 \vee \sim x_0 \vee \sim x_1) \wedge$
$(\sim y_1 \vee x_0 \vee x_1) \wedge$
$(y_1 \vee x_0 \vee \sim x_1) \wedge$
$(y_1 \vee \sim x_0 \vee x_1)$
$\wedge$
$(\sim y_2 \vee x_0 \vee x_1 \vee x_2) \wedge$
$(\sim y_2 \vee \sim x_0 \vee x_1 \vee \sim x_2) \wedge$
$(\sim y_2 \vee x_0 \vee \sim x_1 \vee \sim x_2) \wedge$
$(\sim y_2 \vee \sim x_0 \vee \sim x_1 \vee \sim x_2) \wedge$
$(y_2 \vee \sim x_0 \vee x_1 \vee x_2) \wedge$
$(y_2 \vee x_0 \vee \sim x_1 \vee x_2) \wedge$

$(y_2 \vee \neg x_0 \vee \neg x_1 \vee x_2) \wedge$
$(y_2 \vee x_0 \vee x_1 \vee \neg x_2)$
$\wedge$
$(\neg y_3 \vee x_3 \vee x_2 \vee x_1 \vee x_0) \wedge$
$(\neg y_3 \vee \neg x_3 \vee x_2 \vee x_1 \vee \neg x_0) \wedge$
$(\neg y_3 \vee \neg x_3 \vee x_2 \vee \neg x_1 \vee x_0) \wedge$
$(\neg y_3 \vee \neg x_3 \vee x_2 \vee \neg x_1 \vee \neg x_0) \wedge$
$(\neg y_3 \vee \neg x_3 \vee \neg x_2 \vee x_1 \vee x_0) \wedge$
$(\neg y_3 \vee \neg x_3 \vee \neg x_2 \vee x_1 \vee \neg x_0) \wedge$
$(\neg y_3 \vee \neg x_3 \vee \neg x_2 \vee \neg x_1 \vee x_0) \wedge$
$(\neg y_3 \vee \neg x_3 \vee \neg x_2 \vee \neg x_1 \vee \neg x_0) \wedge$
$(y_3 \vee x_3 \vee x_2 \vee x_1 \vee \neg x_0) \wedge$
$(y_3 \vee x_3 \vee x_2 \vee \neg x_1 \vee x_0) \wedge$
$(y_3 \vee x_3 \vee x_2 \vee \neg x_1 \vee \neg x_0) \wedge$
$(y_3 \vee x_3 \vee \neg x_2 \vee x_1 \vee x_0) \wedge$
$(y_3 \vee x_3 \vee \neg x_2 \vee x_1 \vee \neg x_0) \wedge$
$(y_3 \vee x_3 \vee \neg x_2 \vee \neg x_1 \vee x_0) \wedge$
$(y_3 \vee x_3 \vee \neg x_2 \vee \neg x_1 \vee \neg x_0) \wedge$
$(y_3 \vee \neg x_3 \vee x_2 \vee x_1 \vee x_0)$
$\wedge$
$(\neg y_0 \vee z_0) \wedge$
$(y_0 \vee \neg z_0)$
$\wedge$
$(\neg z_1 \vee \neg y_0 \vee \neg y_1) \wedge$
$(\neg z_1 \vee y_0 \vee y_1) \wedge$
$(z_1 \vee y_0 \vee \neg y_1) \wedge$
$(z_1 \vee \neg y_0 \vee y_1)$
$\wedge$
$(\neg z_2 \vee y_0 \vee y_1 \vee y_2) \wedge$
$(\neg z_2 \vee \neg y_0 \vee y_1 \vee \neg y_2) \wedge$
$(\neg z_2 \vee y_0 \vee \neg y_1 \vee \neg y_2) \wedge$
$(\neg z_2 \vee \neg y_0 \vee \neg y_1 \vee \neg y_2) \wedge$
$(z_2 \vee \neg y_0 \vee y_1 \vee y_2) \wedge$
$(z_2 \vee y_0 \vee \neg y_1 \vee y_2) \wedge$
$(z_2 \vee \neg y_0 \vee \neg y_1 \vee y_2) \wedge$
$(z_2 \vee y_0 \vee y_1 \vee \neg y_2)$
$\wedge$
$(\neg z_3 \vee y_3 \vee y_2 \vee y_1 \vee y_0) \wedge$
$(\neg z_3 \vee \neg y_3 \vee y_2 \vee y_1 \vee \neg y_0) \wedge$
$(\neg z_3 \vee \neg y_3 \vee y_2 \vee \neg y_1 \vee y_0) \wedge$
$(\neg z_3 \vee \neg y_3 \vee y_2 \vee \neg y_1 \vee \neg y_0) \wedge$
$(\neg z_3 \vee \neg y_3 \vee \neg y_2 \vee y_1 \vee y_0) \wedge$
$(\neg z_3 \vee \neg y_3 \vee \neg y_2 \vee y_1 \vee \neg y_0) \wedge$
$(\neg z_3 \vee \neg y_3 \vee \neg y_2 \vee \neg y_1 \vee y_0) \wedge$
$(\neg z_3 \vee \neg y_3 \vee \neg y_2 \vee \neg y_1 \vee \neg y_0) \wedge$
$(z_3 \vee y_3 \vee y_2 \vee y_1 \vee \neg y_0) \wedge$

$(z_3 \vee y_3 \vee y_2 \vee \neg y_1 \vee y_0) \wedge$
$(z_3 \vee y_3 \vee y_2 \vee \neg y_1 \vee \neg y_0) \wedge$
$(z_3 \vee y_3 \vee \neg y_2 \vee y_1 \vee y_0) \wedge$
$(z_3 \vee y_3 \vee \neg y_2 \vee y_1 \vee \neg y_0) \wedge$
$(z_3 \vee y_3 \vee \neg y_2 \vee \neg y_1 \vee y_0) \wedge$
$(z_3 \vee y_3 \vee \neg y_2 \vee \neg y_1 \vee \neg y_0) \wedge$
$(z_3 \vee \neg y_3 \vee y_2 \vee y_1 \vee y_0)$
$\wedge$
$(u_0 \vee u_1 \vee u_2 \vee u_3) \wedge$
$(\neg u_0 \vee x_0 \vee z_0) \wedge$
$(\neg u_0 \vee \neg z_0 \vee \neg x_0) \wedge$
$(\neg x_0 \vee z_0 \vee u_0) \wedge$
$(\neg z_0 \vee x_0 \vee u_0) \wedge$
$(\neg u_1 \vee x_1 \vee z_1) \wedge$
$(\neg u_1 \vee \neg z_1 \vee \neg x_1) \wedge$
$(\neg x_1 \vee z_1 \vee u_1) \wedge$
$(\neg z_1 \vee x_1 \vee u_1) \wedge$
$(\neg u_2 \vee x_2 \vee z_2) \wedge$
$(\neg u_2 \vee \neg z_2 \vee \neg x_2) \wedge$
$(\neg x_2 \vee z_2 \vee u_2) \wedge$
$(\neg z_2 \vee x_2 \vee u_2) \wedge$
$(\neg u_3 \vee x_3 \vee z_3) \wedge$
$(\neg u_3 \vee \neg z_3 \vee \neg x_3) \wedge$
$(\neg x_3 \vee z_3 \vee u_3) \wedge$
$(\neg z_3 \vee x_3 \vee u_3)$

Ukupan broj promenljivih je 16 (po 4 za *x*, *y*, *z* i 4 za smenu *u*), a ukupan broj klauza je (2+4+8+16)*2+17 = 77. Sadržaj pratećeg DIMACS fajla je:

c kodiranje promenljivih ide -> x=1,4; y=5,8; z=9,12; u=13,16;
p cnf 16 77
-1 5 0
1 -5 0

-6 -1 -2 0
-6 1 2 0
6 1 -2 0
6 -1 2 0

-7 1 2 3 0
-7 -1 2 -3 0
-7 1 -2 -3 0
-7 -1 -2 -3 0
7 -1 2 3 0
7 1 -2 3 0
7 -1 -2 3 0
7 1 2 -3 0

-8 4 3 2 1 0
-8 -4 3 2 -1 0
-8 -4 3 -2 1 0
-8 -4 3 -2 -1 0
-8 -4 -3 2 1 0
-8 -4 -3 2 -1 0
-8 -4 -3 -2 1 0
-8 -4 -3 -2 -1 0
8 4 3 2 -1 0
8 4 3 -2 1 0
8 4 3 -2 -1 0
8 4 -3 2 1 0
8 4 -3 2 -1 0
8 4 -3 -2 1 0
8 4 -3 -2 -1 0
8 -4 3 2 1 0

-5 9 0
5 -9 0

-10 -5 -6 0
-10 5 6 0
10 5 -6 0
10 -5 6 0

-11 5 6 7 0
-11 -5 6 -7 0
-11 5 -6 -7 0
-11 -5 -6 -7 0
11 -5 6 7 0
11 5 -6 7 0
11 -5 -6 7 0
11 5 6 -7 0

-12 8 7 6 5 0
-12 -8 7 6 -5 0
-12 -8 7 -6 5 0
-12 -8 7 -6 -5 0
-12 -8 -7 6 5 0
-12 -8 -7 6 -5 0
-12 -8 -7 -6 5 0
-12 -8 -7 -6 -5 0
12 8 7 6 -5 0
12 8 7 -6 5 0
12 8 7 -6 -5 0
12 8 -7 6 5 0

12 8 -7 6 -5 0
12 8 -7 -6 5 0
12 8 -7 -6 -5 0
12 -8 7 6 5 0

13 14 15 16 0
-13 1 9 0
-13 -9 -1 0
-1 9 13 0
-9 1 13 0
-14 2 10 0
-14 -10 -2 0
-2 10 14 0
-10 2 14 0
-15 3 11 0
-15 -11 -3 0
-3 11 15 0
-11 3 15 0
-16 4 12 0
-16 -12 -4 0
-4 12 16 0
-12 4 16 0

Iako je na početku ove sekcije bilo reči o tome kako univerzalan postupak za kodiranje problema u SAT formulu ne postoji iz prethodnih primera možemo uočiti neke smernice. Često se radi o pokazivanju da je formula oblika P => Q tautologija. Potformula P je najčešće formirana od veza između bitova odnosno od prelaska iz jedno stanje u drugo dok je Q invarijanta koju dokazujemo na osnovu veza iz P (prostim jezikom rečeno - Q je tvrdnja do koje nam je stalo). Tautologičnost pokazujemo tako što dokažemo da je negacija formule kontradikcija (ne postoji model za nju). S obzirom na to, imamo situaciju da je formula koju posmatramo ~(P => Q) -> P \/ ~Q idealno nam je da P formiramo da bude KNF, a Q da bude DNF (jer će onda ~Q biti KNF).

Što se samog kodiranja tiče, jedan zgodan pristup je prikazan u primeru 2. Možemo da ispišemo zavisnost rezultujućeg bita od svih mogućih stanja i odatle dođemo do kanonske KNF odnosno kanonske DNF. Tada dobijamo dve ekvivalencije (*b* je bit koji nas interesuje):

b <=> *kanonska KNF od stanja bitova*

b <=> *kanonska DNF od stanja bitova*

Recimo da razbijamo gornju ekvivalenciju, dobili bismo nešto ovako:

(~b \/ *kanonska KNF od stanja bitova*) /\ (b \/ ~(*kanonska KNF od stanja bitova*))

Desni konjunkt nam ne odgovara jer se negacijom KNF dobija DNF i onda je potrebno *“izmnožiti”* levi desni konjukt čime se dobija mnogo klauza. Tu iskoristimo činjenicu da važi:

kanonska DNF od stanja bitova <=> *kanonska KNF od stanja bitova*

i na osnovu teoreme o zameni imamo pravo na izmenu formule u:

(~b \/ *kanonska KNF od stanja bitova*) /\ (b \/ ~(*kanonska DNF od stanja bitova*))

Negacijom DNF se dobija KNF i onda praktično literal *~b*, odnosno literal *b*, se samo dopišu svakom konjunktu kanonske KNF odnosno negacije kanonske DNF.

Postoje nekad situacije gde na potformule P i Q nije moguće primeniti prethodno opisano kodiranje. Tada imamo dva slučaja, ako je potformula dovoljno jednostavno primenimo direktne transformacije,

dok čim se u potformuli javlja veći broj ekvivalencija (dve ili više), najzgodnije je primeniti Cajtinovu transformaciju.

Kombinatorni problemi

Pod kombinatornim problemima, misli se na problem gde postoje osobe A, B i C koje na primer žive u kućama boje P, Q i R i imaju ljubimce S i T. Na osnovu par rečenica koje su najčešće fokusirane na to koja od osoba šta nema, potrebno je odrediti koja osoba je vlasnik životinje T. Dakle, više osoba, zatim uslovi i na kraju upit za specifičnom pripadnošću neke stvari, odnosno nekog bića ili veštine, jednoj od osoba.

Primer 1: Tri đaka Miloš, Zoran i Petar uče strane jezike: engleski i nemački. Ako se zna da bar dvojica govore engleski, i bar dvojica govore nemački, tada postoji učenik koji govori oba jezika. Predstaviti navedenu rečenicu u obliku iskazne formule, a zatim dokazati da je ona tautologija pomoću SAT rešavača. Kodirati u DIMACS formatu i pokrenuti *minisat* rešavač.

Rešenje:

Jedan pristup rešavanju ovog problema može da bude kodiranje gde mi sa p_{ij} kodiramo da i-ti đak zna j-ti jezik. Dakle, ukupno imamo 3 đaka * 2 jezika = 6 promenljivih. Pozabavimo se sada uslovima: *bar dvojica govore engleski*; *bar dvojica govore nemački*. Ovu vrstu pripadnosti izražamo konjunkcijom, a više različitih slučajeva vezujemo disjunkcijama. Detaljnije, ako numerišemo Miloš=1, Zoran=2 i Petar=3 pri čemu engleski=1 i nemački=2, neke mogućnosti su:

- $p_{11} /\backslash p_{21}$, $p_{12} /\backslash p_{32}$ - Miloš i Zoran govore engleski, a Miloš i Petar nemački
- $P_{21} /\backslash p_{31}$, $p_{22} /\backslash p_{32}$ - Zoran i Petar govore oba jezika
- $P_{11} /\backslash p_{31}$, $p_{22} /\backslash p_{32}$ - Miloš i Petar govore engleski, a Zoran i Petar nemački

Potrebno je pokriti sve moguće kombinacije. Uslov da bar dvojica slušaju jezik se prevodi u biranje 2 od 3 (dva đaka od tri) i imamo dva puta takav izbor (jednom za nemački, jednom za engleski) što znači da imamo 3+3=6 opcija:

$((p_{11} /\backslash p_{21}) \backslash/ (p_{11} /\backslash p_{31}) \backslash/ (p_{21} /\backslash p_{31}))$ **/** $((p_{12} /\backslash p_{22}) \backslash/ (p_{12} /\backslash p_{32}) \backslash/ (p_{22} /\backslash p_{32}))$

Levi konjunkt označava da bar 2 đaka slušaju engleski, a desni da bar 2 đaka slušaju nemački.

Treba dokazati da jedan đak zna dva jezika, koristeći isto kodiranje to zapisujemo:

$((p_{11} /\backslash p_{12}) \backslash/$ -> Miloš zna i engleski i nemački

$(p_{21} /\backslash p_{23}) \backslash/$ -> ili Zoran zna i engleski i nemački

$(p_{31} /\backslash p_{32}))$ -> ili Petar zna i engleski i nemački

Dokazivanje da je tautologija se opet svodi na ~(P => Q) -> P \/ ~Q što nam daje:

$(p_{11} \backslash/ p_{21}) /\backslash$

$(p_{11} \backslash/ p_{31}) /\backslash$

$(p_{11} \backslash/ p_{31} \backslash/ p_{21}) /\backslash$

$(p_{21} \backslash/ p_{31})$

/\

$(p_{12} \backslash/ p_{22}) /\backslash$

$(p_{12} \backslash/ p_{32}) /\backslash$

$(p_{12} \backslash/ p_{32} \backslash/ p_{22}) /\backslash$

$(p_{22} \backslash/ p_{32})$

/\

$(\sim p_{11} \backslash/ \sim p_{12}) /\backslash$

$(\sim p_{21} \vee \sim p_{22}) \wedge$
$(\sim p_{31} \vee \sim p_{32})$

Ukupno 6 promenljivih i 11 klauza. Odgovarajuća DIMACS datoteka:

c p11 p12 p21 p22 p31 p32
c 1 2 3 4 5 6
p cnf 6 11
1 3 0
1 5 0
1 5 3 0
3 5 0
2 4 0
2 6 0
2 6 4 0
4 6 0
-1 -2 0
-3 -4 0
-5 -6 0

Primer 2: Četiri čoveka nose prezimena Smith, Baker, Carpenter i Tailor i imaju neku od četiri profesije: kovač (eng. *smith*), pekar (eng. *baker*), stolar (eng. *carpenter*) i krojač (eng. *tailor*), ali ni jedan od njih nema profesiju koja je sadržana u njegovom prezimenu. Takođe, svaki od njih ima po jednog sina koji se takođe bavi nekom od ove četiri profesije, ali takođe niko onom profesijom koja je sadržana u njegovom prezimenu. Niko od sinova nema istu profesiju kao svoj otac. Baker ima istu profesiju kao Carpenter-ov sin, dok je Smith-ov sin pekar. Pronaći profesije očeva i sinova.

Rešenje:

Treba kodirate sledeće informacije:

1. Svaki čovek se bavi bar jednom profesijom
2. Nijedan čovek se ne bavi sa više profesija
3. Niko se ne bavi profesijom istog naziva kao sopstveno prezime
4. Ničiji sin se ne bavi istom profesijom kao i otac
5. Date veze između profesija konkretnih ljudi

Kao i u primeru 1, možemo koristiti slično kodiranje, x_{ij} označava da se i-ti otac bavi j-tom profesijom, dok sa y_{ij} obeležavamo da se i-ti sin bavi j-tom profesijom. Ako kodiramo ljude i njihove sinove redom kojim se pojavljuju u tesktu (Smith=x_{1*}, Smith-ov sin=y_{1*} itd.), isto i za profesije (x_{*1} označava da je neko kovač), tada za stavku 1 imamo uslove (4*2=8 klauza):

$(x_{11} \vee x_{12} \vee x_{13} \vee x_{14}) \wedge$ -> Smith je ili kovač, ili pekar, ili stolar, ili krojač
$(x_{21} \vee x_{22} \vee x_{23} \vee x_{24}) \wedge$
$(x_{31} \vee x_{32} \vee x_{33} \vee x_{34}) \wedge$
$(x_{41} \vee x_{42} \vee x_{43} \vee x_{44})$
$\wedge$
$(y_{11} \vee y_{12} \vee y_{13} \vee y_{14}) \wedge$ -> Smith-ov sin je ili kovač, ili pekar, ili stolar, ili krojač
$(y_{21} \vee y_{22} \vee y_{23} \vee y_{24}) \wedge$
$(y_{31} \vee y_{32} \vee y_{33} \vee y_{34}) \wedge$
$(y_{41} \vee y_{42} \vee y_{43} \vee y_{44})$

Za stavku dva, posmatrajmo jednu osobu. Uslov je da ne postoje dve različite profesije kojima se osoba bavi. S obzirom na to da imamo četiri profesije od kojih biramo dve, imamo $\binom{4}{2}$ različitih parova tj. 6 klauza za jednu osobu (ukupno 8*6=48 klauza):

$(\sim x_{11} \vee \sim x_{12}) \wedge$
$(\sim x_{11} \vee \sim x_{13}) \wedge$
$(\sim x_{11} \vee \sim x_{14}) \wedge$
$(\sim x_{12} \vee \sim x_{13}) \wedge$
$(\sim x_{12} \vee \sim x_{14}) \wedge$
$(\sim x_{13} \vee \sim x_{14}) \wedge$
$(\sim x_{21} \vee \sim x_{22}) \wedge$
$(\sim x_{21} \vee \sim x_{23}) \wedge$
$(\sim x_{21} \vee \sim x_{24}) \wedge$
$(\sim x_{22} \vee \sim x_{23}) \wedge$
$(\sim x_{22} \vee \sim x_{24}) \wedge$
$(\sim x_{23} \vee \sim x_{24}) \wedge$
$(\sim x_{31} \vee \sim x_{32}) \wedge$
$(\sim x_{31} \vee \sim x_{33}) \wedge$
$(\sim x_{31} \vee \sim x_{34}) \wedge$
$(\sim x_{32} \vee \sim x_{33}) \wedge$
$(\sim x_{32} \vee \sim x_{34}) \wedge$
$(\sim x_{33} \vee \sim x_{34}) \wedge$
$(\sim x_{41} \vee \sim x_{42}) \wedge$
$(\sim x_{41} \vee \sim x_{43}) \wedge$
$(\sim x_{41} \vee \sim x_{44}) \wedge$
$(\sim x_{42} \vee \sim x_{43}) \wedge$
$(\sim x_{42} \vee \sim x_{44}) \wedge$
$(\sim x_{43} \vee \sim x_{44})$
$\wedge$
$(\sim y_{11} \vee \sim y_{12}) \wedge$
$(\sim y_{11} \vee \sim y_{13}) \wedge$
$(\sim y_{11} \vee \sim y_{14}) \wedge$
$(\sim y_{12} \vee \sim y_{13}) \wedge$
$(\sim y_{12} \vee \sim y_{14}) \wedge$
$(\sim y_{13} \vee \sim y_{14}) \wedge$
$(\sim y_{21} \vee \sim y_{22}) \wedge$
$(\sim y_{21} \vee \sim y_{23}) \wedge$
$(\sim y_{21} \vee \sim y_{24}) \wedge$
$(\sim y_{22} \vee \sim y_{23}) \wedge$
$(\sim y_{22} \vee \sim y_{24}) \wedge$
$(\sim y_{23} \vee \sim y_{24}) \wedge$
$(\sim y_{31} \vee \sim y_{32}) \wedge$
$(\sim y_{31} \vee \sim y_{33}) \wedge$
$(\sim y_{31} \vee \sim y_{34}) \wedge$
$(\sim y_{32} \vee \sim y_{33}) \wedge$
$(\sim y_{32} \vee \sim y_{34}) \wedge$
$(\sim y_{33} \vee \sim y_{34}) \wedge$
$(\sim y_{41} \vee \sim y_{42}) \wedge$
$(\sim y_{41} \vee \sim y_{43}) \wedge$
$(\sim y_{41} \vee \sim y_{44}) \wedge$

$(\sim y_{42} \vee \sim y_{43}) \wedge$
$(\sim y_{42} \vee \sim y_{44}) \wedge$
$(\sim y_{43} \vee \sim y_{44})$

Stavka tri, a to je da se nijedna osoba ne bavi profesijom koja prezimenu te osobe, se jednostavno kodira jediničnim klauzama (8 klauza):

$\sim x_{11} \wedge \sim x_{22} \wedge \sim x_{33} \wedge \sim x_{44}$
$\wedge$
$\sim y_{11} \wedge \sim y_{22} \wedge \sim y_{33} \wedge \sim y_{44}$

Da se sinovi ne bave istom profesijom kao i očevi kodiramo negiranjem da se bave istom profesijom. Dakle za svaku profesiju imamo po četiri ograničenja, ili ekvivalentno za svaki par otac-sin imamo četiri ograničenja koja su određena sa četiri profesije (4*4=16 klauza):

$(\sim x_{11} \vee \sim y_{11}) \wedge$
$(\sim x_{12} \vee \sim y_{12}) \wedge$
$(\sim x_{13} \vee \sim y_{13}) \wedge$
$(\sim x_{14} \vee \sim y_{14}) \wedge$
$(\sim x_{21} \vee \sim y_{21}) \wedge$
$(\sim x_{22} \vee \sim y_{22}) \wedge$
$(\sim x_{23} \vee \sim y_{23}) \wedge$
$(\sim x_{24} \vee \sim y_{24}) \wedge$
$(\sim x_{31} \vee \sim y_{31}) \wedge$
$(\sim x_{32} \vee \sim y_{32}) \wedge$
$(\sim x_{33} \vee \sim y_{33}) \wedge$
$(\sim x_{34} \vee \sim y_{34}) \wedge$
$(\sim x_{41} \vee \sim y_{41}) \wedge$
$(\sim x_{42} \vee \sim y_{42}) \wedge$
$(\sim x_{43} \vee \sim y_{43}) \wedge$
$(\sim x_{44} \vee \sim y_{44})$

Poslednja grupa ograničenja se odnosi na konkretne osobe i konkretne profesije. Počnimo od jednostavnijeg uslova, *Smith-ov sin je pekar* kodiramo sa jednom klauzom:

y_{12}

Baker se bavi istom profesijom kao Carpenter-ov sin kodiramo sa četiri disjunkcije (zajedno su ili kovači, ili pekari, ili stolari, ili krojači):

$(x_{21} \wedge y_{31}) \vee (x_{22} \wedge y_{32}) \vee (x_{23} \wedge y_{33}) \vee (x_{24} \wedge y_{34})$

Kada ove disjunkte "izmnožimo" dobijemo sledeće klauze (njih 16):

$(x_{21} \vee x_{22} \vee x_{23} \vee x_{24}) \wedge$
$(x_{21} \vee x_{22} \vee x_{23} \vee y_{34}) \wedge$
$(x_{21} \vee x_{22} \vee y_{33} \vee x_{24}) \wedge$
$(x_{21} \vee x_{22} \vee y_{33} \vee y_{34}) \wedge$
$(x_{21} \vee y_{32} \vee x_{23} \vee x_{24}) \wedge$
$(x_{21} \vee y_{32} \vee x_{23} \vee y_{34}) \wedge$
$(x_{21} \vee y_{32} \vee y_{33} \vee x_{24}) \wedge$
$(x_{21} \vee y_{32} \vee y_{33} \vee y_{34}) \wedge$
$(y_{31} \vee x_{22} \vee x_{23} \vee x_{24}) \wedge$
$(y_{31} \vee x_{22} \vee x_{23} \vee y_{34}) \wedge$
$(y_{31} \vee x_{22} \vee y_{33} \vee x_{24}) \wedge$
$(y_{31} \vee x_{22} \vee y_{33} \vee y_{34}) \wedge$

(y_{31} \/ y_{32} \/ x_{23} \/ x_{24}) /\
(y_{31} \/ y_{32} \/ x_{23} \/ y_{34}) /\
(y_{31} \/ y_{32} \/ y_{33} \/ x_{24}) /\
(y_{31} \/ y_{32} \/ y_{33} \/ y_{34})

Kada se sve do sada navedene klauze, tj. njih 97 kodiraju u DIMACS cnf format dobija se sledeći sadržaj datoteke (kodirane klauze nisu u redosledu izlaganja):

c x_11 x_12 x_13 x_14 x_21 x_22 x_23 x_24 x_31 x_32 x_33 x_34 x_41 x_42 x_43 x_44
c 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
c y_11 y_12 y_13 y_14 y_21 y_22 y_23 y_24 y_31 y_32 y_33 y_34 y_41 y_42 y_43 y_44
c 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32

p cnf 32 97
1 2 3 4 0
5 6 7 8 0
9 10 11 12 0
13 14 15 16 0
-1 -2 0
-1 -3 0
-1 -4 0
-2 -3 0
-2 -4 0
-3 -4 0
-5 -6 0
-5 -7 0
-5 -8 0
-6 -7 0
-6 -8 0
-7 -8 0
-9 -10 0
-9 -11 0
-9 -12 0
-10 -11 0
-10 -12 0
-11 -12 0
-13 -14 0
-13 -15 0
-13 -16 0
-14 -15 0
-14 -16 0
-15 -16 0
-1 0
-6 0
-11 0
-16 0
17 18 19 20 0
21 22 23 24 0

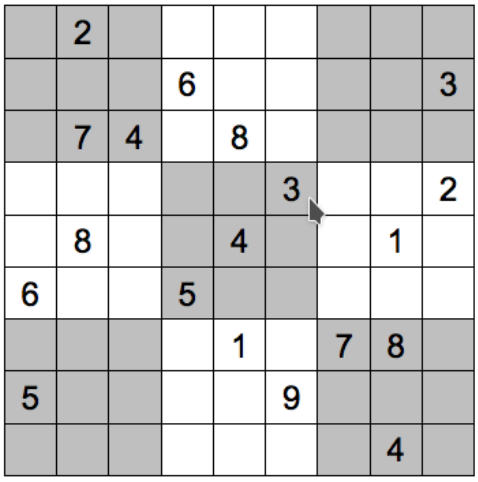
25 26 27 28 0
29 30 31 32 0
-17 -18 0
-17 -19 0
-17 -20 0
-18 -19 0
-18 -20 0
-19 -20 0
-21 -22 0
-21 -23 0
-21 -24 0
-22 -23 0
-22 -24 0
-23 -24 0
-25 -26 0
-25 -27 0
-25 -28 0
-26 -27 0
-26 -28 0
-27 -28 0
-29 -30 0
-29 -31 0
-29 -32 0
-30 -31 0
-30 -32 0
-31 -32 0
-17 0
-22 0
-27 0
-32 0
-1 -17 0
-2 -18 0
-3 -19 0
-4 -20 0
-5 -21 0
-6 -22 0
-7 -23 0
-8 -24 0
-9 -25 0
-10 -26 0
-11 -27 0
-12 -28 0
-13 -29 0
-14 -30 0
-15 -31 0
-16 -32 0

5 6 7 8 0
5 6 7 28 0
5 6 27 8 0
5 6 27 28 0
5 26 7 8 0
5 26 7 28 0
5 26 27 8 0
5 26 27 28 0
25 6 7 8 0
25 6 7 28 0
25 6 27 8 0
25 6 27 28 0
25 26 7 8 0
25 26 7 28 0
25 26 27 8 0
25 26 27 28 0
18 0

Primetimo da i kod ovog tipa problema postoje određene pravilnost odnosno smernice za kodiranje uslova datih u tekstu. Da označimo da entitet *a* može imati pridružen entitet *b* uvodimo promenljive sa duplim indeksiranjem. Na primer, sa x_{ij} smo u drugom primeru označavali da se i-ti ovek bavi j-tom profesijom, a u prvom primeru da i-ti đak zna j-ti jezik. S druge strane kodiranje samih uslova nije jednostavno pretočiti u šablon. Ako imamo pozitivne uslove sa više ishoda, na primer *bar jedan đak zna dva jezika*, to kodiramo disjunkcijom konjunkcija (prvi đak zna prvi **i** drugi jezik **ili** drugi đak zna prvi **i** drugi jezik itd.). S druge strane, ako imamo negativne uslove u pitanju je konjunkcija više disjunkcija. To je prirodno jer ona može da se gleda kao negacija pozitivnog ishoda pri čemu konjukcije pređu u disjunkcije i obrnuto. Na kraju konkretne uslove je potrebno zapisati po potrebi i nema nekog opšteg pravila koje se može primeniti.

Primer rešavanja logičkih igara - sudoku

Sudoku je jednostavna igra sa brojevima na kvadratnoj mreži. Osnovna ideja je da treba popuniti nedostajuće vrednosti na kvadratnoj mreži tako da se ni u jednom redu, koloni ili kvadratu ne nađu dva ista broja. Kada se kaže samo sudoku podrazumeva se mreža od 81-jednog polja i cifre od 1 do 9, na primer:



Na početku su popunjena samo neka polja, igrač treba da dopiše vrednosti ostalih kvadratića na taj način da se zadovolje navedeni uslovi.

U opštem slučaju posmatramo tabelu dimenzija NxN gde N za predstavljeni primer ima vrednost 9. Postavlja se pitanje kako kodirati da polje indeksirano sa *i*, *j* ima vrednost *k* na sebi? Radi ilustracije, na sudoku mreži datoj malo iznad ovog pasusa, polje sa indeksom 1, 3 ima vrednost šest. Za svako polje potrebno je kodirati N vrednosti. Koristićemo promenljive $x_{i,j,k}$ pri čemu je *i* indeks reda, *j* indeks kolone i na kraju *k* je vednost dodeljena tom polju. Ako polje sa indeksima i,j ima vrednost *k* tada je promenljiva $x_{i,j,k}$ tačna, a netačna inače.

Uslovi koje želimo da zapišemo uz pomoć ovih promenljivih su (prva dva uslova su generički, a ostali su sudoku specifični):

1. Svako polje mora da ima vrednost u rasponu od 1 do N:
   $x_{i,j,1} \vee x_{i,j,2} \vee x_{i,j,3} \vee \ldots \vee x_{i,j,N}$ -> za konkretno polje *i,j* mora da važi da je postavljena jedna od ovih vrednosti; imamo po jednu ovakvu klauzu za svako polje.
2. Nijedno polje ne može da ima dve pridružene vrednosti:
   $\sim(x_{i,j,k1} \wedge x_{i,j,k2}) \rightarrow \sim x_{i,j,k1} \vee \sim x_{i,j,k2}$ -> za sve parove mogućih vrednosti polja *k1* i *k2* ne sme da se desi da isto polje ima i vrednost *k1* i vrednost *k2*. Broj ovakvih parova je $\binom{N}{2}$, za svako polje imamo broj klauza koji jednak broju parova.
3. Ni u jednom redu ne možemo imati dve iste vrednosti:
   $\sim(x_{i,j1,k} \wedge x_{i,j2k}) \rightarrow \sim x_{i,j1,k} \vee \sim x_{i,j2,k}$ -> za sve parove kolona j1 i j2 fiksnog reda *i* važi da ne mogu imati istu vrednost *k*. Broj ovakvih parova je $\binom{N}{2}$, za svako *i* i svako *k* mi imamo broj klauza jednak broju parova.
4. Ni u jednoj koloni ne možemo imati dve iste vrednosti:
   $\sim(x_{i1,j,k} \wedge x_{i2,j,k}) \rightarrow \sim x_{i1,j,k} \vee \sim x_{i2,j,k}$ -> slično kao za 3. samo što su u pitanju redovi, a ne kolone
5. Ni u jednom kvadratu ne možemo imati dve iste vrednosti:
   $\sim(x_{i1,j1,k} \wedge x_{i2,j2,k}) \rightarrow \sim x_{i1,j1,k} \vee \sim x_{i2,j2,k}$ -> za <u>različita</u> polja *i1,j1* i *i2,j2* koja se nalaze u okviru istog kvadrata važi da ne mogu imati istu vrednost. Imamo N kvadrata koji sadrže po $\binom{N}{2}$ parova polja pri čemu takođe *k* uzima vrednosti od 1 do N.
6. Početne vrednosti sudoku mreže se kodiraju jediničnim klauzama koje odgovaraju polju i vrednosti, na primer za prikazanu sudoku tablu imali bismo $x_{1,2,2} \wedge x_{2,4,6} \wedge x_{2,9,3}$ i tako dalje.

Vidimo da je broj promenljivih reda N^3 dok je broj klauza reda N^4. Ispostavlja se da već za standardno N=9 imamo 729 promenljivih i 11788 klauza (broj klauza će varirati u zavisnosti od inicijalne popunjenosti sudoku tabele). S obzirom na dimenzije problema konkretan sadržaj DIMACS datoteke će biti izostavljen iz ovog dokumenta. Dodatno, trebalo bi primetiti da je pisanje programa

koji generiše CNF za sudoku jedini realističan način da se SAT kodiranje primeni (pri čemu je pisanje ovakvog programa prilično jednostavno). Prilikom samog kodiranja promenljivih u DIMACS formatu koristimo mapiranje $x_{i,j,k}$ -> p_t pri čemu t = i * N^2 + j * N + k.
Prikazano kodiranje problema nije najefikasnije i postoje efikasnija kodiranja (primer jednog je dat na Veb adresi http://www.cs.cmu.edu/~hjain/papers/sudoku-as-SAT.pdf).

Čas 5 - vežbanje za kolokvijum

Preduslovi*: časovi vežbi od prvog do četvrtog;*

Uvod

S prethodnim poglavljem je zaokružena priča o iskaznoj logici koja se svodi na dva dela:

1. Struktura iskaznih formula, osnovne transformacije i algoritmi u programskom jeziku C++
2. Kodiranje raznih problema u instance SAT problema

U nastavku ovog poglavlja slede zadaci za samostalno vežbanje i proveru kako je izložena materija savladana. Iako su rešenja ovih zadataka prisutna u dodatku, čitalac bi **trebalo** da uloži značajan trud da reši ove zadatke sam.

C++ zadaci

1. U programskom jeziku C++:
    a. Definisati strukture podataka za predstavljanje iskaznih formula u obliku sintaksnog stabla. Pretpostaviti da formule sadrže samo veznike /\, \/, ~, =>, kao i da nema logičkih konstanti. Implementirati funkciju *printFormula()* za prikazivanje formule na standardnom izlazu u uobičajenoj konkretnoj sintaksi.
    b. Dužina puta u stablu jednaka je broju grana na tom putu. Visina stabla je jednaka dužini najdužeg puta od korena ka nekom listu u stablu. Visina negacije u formuli *~Formula* je jednaka visini stabla kojim je predstavljena formula *Formula*. NNF-udaljenost formule *Formula* je zbir visina svih negacija u toj formuli. Implementirati metode koje izračunavaju NNF -udaljenost date formule *Formula*.
    c. Napisati program koji kreira iskaznu formulu: *~(x => (y \/ ~z)) /\ ~(~x \/ ~(y => z))*, a zatim izračunava njenu NNF -udaljenost.
2. U programskom jeziku C++:
    a. Definisati strukture podataka za predstavljanje iskaznih formula u obliku sintaksnog stabla. Pretpostaviti da formule sadrže samo veznike /\, \/, ~, =>, kao i da nema logičkih konstanti. Implementirati funkciju *printFormula()* za prikazivanje formule na standardnom izlazu u uobičajenoj konkretnoj sintaksi.
    b. Napisati funkciju koja proverava da li je formula *Formula* logička posledica konačnog skupa formula D, tj. da li svaka valuacija koja zadovoljava sve formule iz D zadovoljava i formulu *Formula*.
    c. Napisati funkciju koja proverava da li je konačan skup formula D nezavisan skup formula, tj. da li za svaku formulu *Formula* iz D ne važi da je *Formula* logička posledica skupa D \ {*Formula*}.

d. Napisati funkciju koja za dati skup formula D pronalazi minimalni podskup D_0 takav da je istovremeno nezavisan i ekvivalentan sa D. Skupovi D i D_0 su ekvivalentni ako za svaku formulu *Formula* koja je logička posledica skupa D, važi da je *Formula* logička posledica skupa D_0 i obratno (primetiti da ako je *Formula* iz D, i logička posledica skupa D \ {*Formula*}, tada je D ekvivalentno sa D \ {F }).
e. Napisati program koji kreira formule: *~p \/ q, ~(r => s) => (p => q), ~q => ~p* i za taj skup formula određuje minimalni ekvivalentni podskup.

3. U programskom jeziku C++:
   a. Omogućiti predstavljanje CNF iskaznih formula (u obliku skupa skupova literala, ili nekako drugačije). Omogućiti prikaz na standardnom izlazu.
   b. Implementirati funkciju *resolve(c1, c2, p, r)* koja primenjuje pravilo rezolucije nad klauzama *c1* i *c2* po literalu *p* i popunjava rezolventu *r* (pretpostavka je da *c1* sadrži *p*, a *c2* sadrži ~p). Ovim pravilom se dobija nova klauza *r* koja sadrži sve literale iz obe polazne klauze, ali bez literala *p* iz *c1* i literala *~p* iz *c2* (tj. *r = (c1 \ p) U (c2 \ ~p)*). Funkcija treba da eliminiše duplikate iz dobijene klauze. Ukoliko je novodobijena klauza tautologija (tj. sadrži i neko iskazno slovo *q* i njegovu negaciju), tada funkcija vraća *false*, a u suprotnom vraća *true*.
   c. Implementirati funkciju *eliminate(f, p)* koja uklanja iskazno slovo *p* iz CNF formule *f* tako što primenjuje pravilo rezolucije (koristeći funkciju resolve iz dela pod *b.* nad svakim parom klauza (*c1*, *c2*) takvim da je p element *c1* i *~p* element *c2*. Klauze koje se dobiju na ovaj način se dodaju u CNF formulu, a originalne klauze (dakle, sve one koje sadrže bilo *p* bilo *~p*) se brišu. Ostale klauze iz originalne formule (one koje ne sadrže ni *p* ni *~p*) ostaju u CNF formuli nepromenjene. Ako se rezolucijom dobije tautologija, tada se ta klauza ne dodaje u CNF formulu. Ako se rezolucijom dobije prazna klauza, funkcija vraća *false*, a u suprotnom vraća *true*.
   d. Implementirati funkciju *checkSat(f)* koja za datu CNF formulu *f* eliminiše jednu po jednu promenljivu, koristeći funkciju *eliminate()* iz dela pod *c.*. Ako bilo koji poziv funkcije *eliminate()* vrati *false*, tada i *checkSat()* vraća *false* (formula je nezadovoljiva jer je izvedena prazna klauza). Ako se proces završi bez izvođenja prazne klauze, funkcija vraća *true* (formula je zadovoljiva).
   e. Napisati program koji kreira CNF formulu: *[[p, ~q, r] , [~p, ~q] , [q, p, ~r] , [q, ~p, ~r]]*, a zatim za nju proverava da li je zadovoljiva, pozivom funkcije *checkSat()* iz dela pod *d.*.

MiniSAT problemi

1. Neka je dat četvorobitni pomerački registar sa linearnom povratnom spregom čiji su bitovi (u i-tom stanju) obelezeni sa p_i, q_i, r_i, s_i, počev od bita najveće težine. Registar u svakom koraku prelazi iz stanja u stanje na sledeći način:
   $p_{i+1} = r_i \wedge s_i$ (^ je ekskluzivna disjunkcija)
   $q_{i+1} = p_i$
   $r_{i+1} = q_i$
   $s_{i+1} = r_i$
   Ako važi da je $(p_4, q_4, r_4, s_4) = (1, 0, 1, 0)$, odrediti početno stanje (p_0, q_0, r_0, s_0). Zadatak rešiti svođenjem na SAT, a dobijenu iskaznu formulu u CNF-u predstaviti u DIMACS formatu i pokrenuti *minisat* rešavač.

2. Dat je trobitni pomerački registar (*p*, *q*, *r*) i bit prenosa *c*. Obavlja se sledeća operacija, najpre se sadržaj registra obrće u ogledalu, zatim se dobijeni sadržaj pomera u desno kroz bit prenosa (što znači da vrednost bita *c* dolazi na najviši bit u registru, a najniži bit iz registra se potiskuje u *c*), nakon čega se sadržaj registra ponovo okrene u ogledalu. Dokazati da se nakon 4 primene ove složene operacije registar i bit prenosa vraćaju u početno stanje.
3. Predstaviti četvorobitni potpuni sabirač iskaznom formulom. Zatim pomoću SAT rešavača odrediti razliku dva četvorobitna broja.
   Pomoć: Neka je prvi sabirak $x_3x_2x_1x_0$, drugi $y_3y_2y_1y_0$ i zbir $z_3z_2z_1z_0$. Konstruisati veze između z_i i x odnosno y za svaki bit iz z. Kada je to urađeno oduzimanje brojeva, na primer 0110 i 0100, se realizuje tako što "tražite" y takvo da za z=0110 i x=0100 važi z=x+y (za primer dodaju se klauze $\sim z_3$, z_2, z_1, $\sim z_0$, $\sim x_3$, x_2, $\sim x_1$, $\sim x_0$, a rezultat se čita iz bitova broja y valuacije koja je model za proširenu formulu).
4. Dat je četvorobitni registar, nad kojim je definisana sledeća operacija, najpre se od sadržaja oduzme 1, a onda se bitovi rezultata komplementiraju. Dokazati da se nakon dve uzastopne primene ove operacije dobija početna vrednost. Zadatak rešiti svođenjem na SAT. Dobijenu iskaznu formulu kodirati u DIMACS formatu, a zatim pokrenuti *minisat* rešavač.

Čas 6 - Logika prvog reda, formiranje formula

Preduslovi: *časovi 1 i 2*;

Motivacija

Na prethodnim časovima vežbi smo videli da je iskazna logika moćan alat za modelovanje raznih problema. Međutim ispostavlja se da je izražajnost iskazne logike nedovoljna za određene vrste problema. Na primer, da li postoji model za:

$$x < y, x-5 < z, y > z$$

tj. takvi *x*, *y* i *z* koji zadovoljavaju gornje izraze? Ili kako rezonovati o tvrđenju da za proizvoljan ceo broj *k* važi da 2┼2*k+1 (2*k+1 nije deljivo sa 2)?
Odgovarajući alat za modelovanje prikazanih izraza, odnosno tvrđenja, predstavlja logika prvog reda. Ona nam upotrebom *kvantifikatora* omogućava da rezonujemo o tome da tvrđenje važi za sve instance, ili da postoji bar jedna instanca za koju tvrđenje važi. Dalje, uvodi funkcijske simbole, na primer x-5 se može gledati i kao **-**(x, 5) gde je **-** simbol funkcije. Slično, omogućava nam da uspostavi odnose između dve instance, na primer jedno značenje izraza *x* < *y* bi mogli da bude "x je manje od y". Tu dolazimo do pojma značenja nekog simbola koji vidimo ispred sebe. Da li je uslov:

$$x >= x + 0.25$$

uvek ispunjen? Ako je izvađen iz konteksta:

```
int x;
if (x >= x + 0.25)
{
        …
}
```

onda smatramo da jeste, dok s druge strane ako je *x* realan broj napisan ispred nas na papiru onda imamo da uslov nikada nije ispunjen. Funkcijski i relacijski simboli trebaju pridruženo značenje da bi bili ispravno tumačeni. Logika prvog reda nam daje da upotrebljavamo kvantifikatore, promenljive,

funkcijske simbole, relacijske simbole i konstante, dok nam *teorije u pozadini* (eng. background theories) omogućavaju da ti simboli dobiju neko smisleno značenje. Ove dve komponente nam zajedno omogućavaju da sprovodimo izuzetno bogato i kompleksno rezonovanje.

Sintaksa logike prvog reda

Def 6.1: Jezik, odnosno signatura $\mathbf{L}$ *čini skup funkcijskih simbola* $\mathbf{Func}$*, skup relacijskih simbola* $\mathbf{Rel}$ *i funkcija* $\mathbf{ar}$*:* $(\mathbf{Func} \cup \mathbf{Rel}) \to \mathbf{N}_0$ *koja svakom simbolu dodeljuje arnost (broj argumenata funkcijskog odnosno relacijskog simbola).*

Funkcijski simboli arnosti nula su konstante, a relacijski simboli arnosti nula su promenljive (drugi naziv za relacijske simbole je i predikatski simboli).

Slično kao u iskaznoj logici, sintaksu predikatske logike definišemo rekurzivno kroz pojmove:

1. Term (intuitivno, odnosi se na kombinacije konstanti i funkcijskih simbola)
2. Atomička formula (intuitivno, odnosi se na primenu relacija na termove)
3. Formula (intuitivno, odnosi se na primenu logičkih veznika i kvantifikatora na atomičke formule, kao i na same atomičke formule)

Def 6.2: Skup termova nad jezikom $\mathbf{L}$ *je najmanji skup koji zadovoljava:*

- *Svaka promenljiva je term*
- *Ako je* $\mathbf{c}$ *simbol konstante jezika* $\mathbf{L}$ *onda je* $\mathbf{c}$ *term*
- *Ako su* $\mathbf{t_1}, \ldots, \mathbf{t_k}$ *termovi i* $\mathbf{f}$ *funkcijski simbol jezika* $\mathbf{L}$ *onda je i* $\mathbf{f}(\mathbf{t_1}, \ldots, \mathbf{t_k})$ *takođe term*

Def 6.3: Atomičke formule jezika $\mathbf{L}$ *su najmanji skup koji zadovoljava da je* $\mathbf{r}$ *relacijski simbol jezika* $\mathbf{L}$ *arnosti* $\mathbf{k}$ *i da su* $\mathbf{t_1}, \ldots, \mathbf{t_k}$ *termi jezika* $\mathbf{L}$*, tada je i* $\mathbf{r}(\mathbf{t_1}, \ldots, \mathbf{t_k})$ *takođe atomička formula.*

Def 6.4: Skup formula logike prvog reda je najmanji skup formula koji zadovoljava:

- *Atomičke formule su formule*
- $\sim(\mathbf{A})$ *je formula ako je* $\mathbf{A}$ *formula*
- *Ako su* $\mathbf{A}$ *i* $\mathbf{B}$ *formule tada su formule i:*
  - $(\mathbf{A}) \wedge (\mathbf{B})$
  - $(\mathbf{A}) \vee (\mathbf{B})$
  - $(\mathbf{A}) \Rightarrow (\mathbf{B})$
  - $(\mathbf{A}) \Leftrightarrow (\mathbf{B})$
- *Ako je* $\mathbf{A}$ *formula, a* $\mathbf{x}$ *promenljiva, tada su formule i:*
  - $\forall x.(\mathbf{A})$
  - $\exists x.(\mathbf{A})$

S obzirom na to da uvođenje kvantifikatora dodatno kompikuje rezonovanje o zadovoljivosti / valjanosti formula definisaćemo još jedan pojma koji se odnosi na to da li neki kvantifikator utiče na promenljivu.

Def 6.5: Svako pojavljivanje promenljive u okviru atomičke formule je slobodno. Sva slobodna pojavljivanja promenljivih u formuli $\mathbf{A}$ *su slobodna i u* $\sim(\mathbf{A})$*. Sva slobodna pojavljivanja promenljivih u formulama* $\mathbf{A}$ *i* $\mathbf{B}$ *su slobodna i u* $(\mathbf{A}) \wedge (\mathbf{B})$*,* $(\mathbf{A}) \vee (\mathbf{B})$*,* $(\mathbf{A}) \Rightarrow (\mathbf{B})$*,* $(\mathbf{A}) \Leftrightarrow (\mathbf{B})$*. Sva slobodna pojavljivanja promenljivih različitih od* $\mathbf{x}$ *u formuli* $\mathbf{A}$ *su slobodna i u* $\forall x.(\mathbf{A})$*,* $\exists x.(\mathbf{A})$*. Sva pojavljivanja promenljivih koja nisu slobodna su vezana (kvantifikatorom).*

Na kraju kratke priče o sintaksi dodajemo još dva pojma koja će nam značiti kasnije kada budemo diskutovali o semantici.

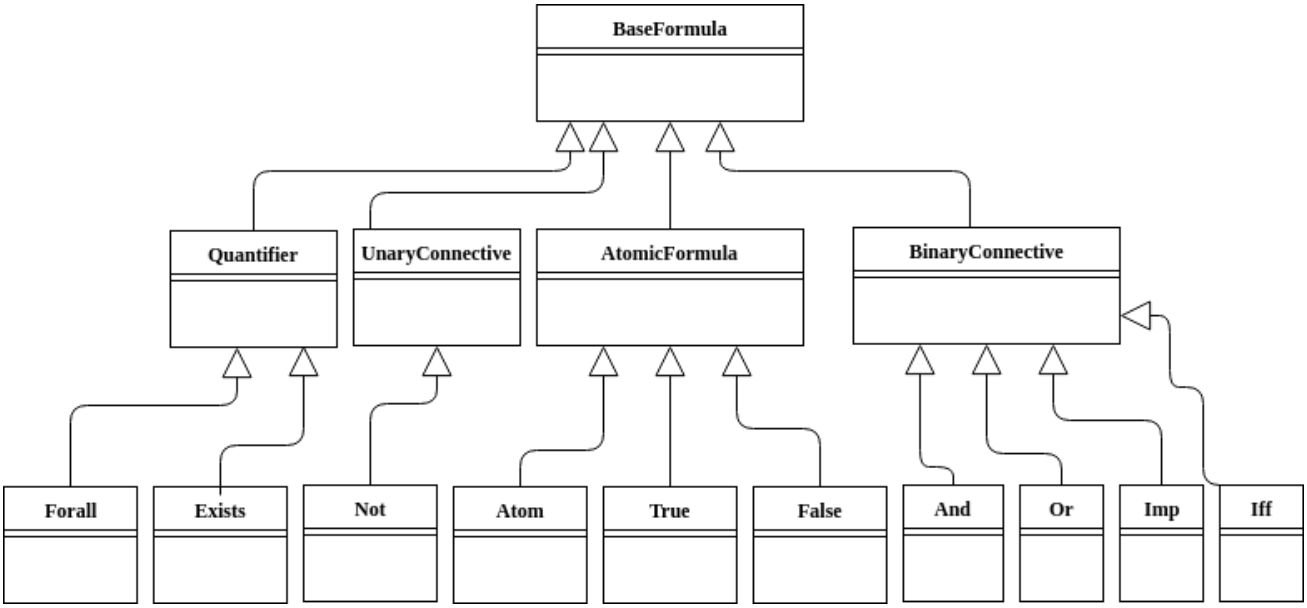
Def 6.6: Formula je rečenica akko nema slobodnih promenljivih. Formula je bazna akko ne sadrži promenljive.

Sintaksa sa stanovišta implementacije

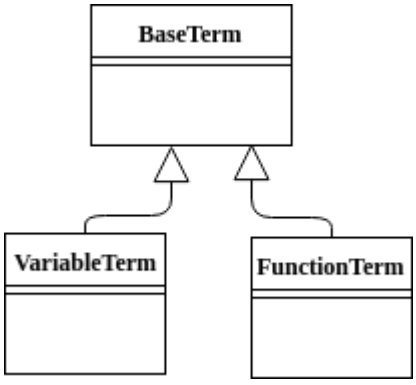
Kada smo implementirali sintaksnu strukturu formula iskazne logike došli smo do hijerarhije klasa date u Hijerarhija klasa za predstavljanje logičkih izraza. Što se same formule tiče, tu nema puno promena. Naravno, potrebno je dodati pojam kvantifikatora odnosno formule koja sadrži kvantifikator. Dakle, klasu *BaseFormula* nasleđuje klasa *Quantifier* koja je ujedno i roditeljska klasa za specifične formule tipa *Forall* i *Exists*. Obe specijalizacije treba da imaju pristup formuli na koju se primenjuje kvantifikator, kao i promenljivoj koja se njime vezuje, pa će to biti izdvojeno u *Quatifier* klasu (dijagram 6.1).

S druge strane imamo celu jednu novu hijerarhiju klasa. Dakle, potrebno je predstaviti termove odgovarajućim tipovima. Direktno iz definicije termova dolazi se do bazne klase *BaseTerm* i izvedenih klasa *FunctionTerm* odnosno *VariableTerm* (dijagram 6.2). Prva je kombinacija funkcijskog simbola i nula ili više pridruženih termova, dok druga sadrži oznaku promenljive.

Naredna stvar kojoj treba posvetiti pažnju je struktura samog atoma. Klasa *Atom* je sada relacijski simbol koji se primenjuje na odgovarajući broj termova i ima malo komplikovaniju sadržinu.



Dijagram 6.1: Hijerarhija klasa za predstavljanje formula logike prvog reda



Dijagram 6.2: Hijerarhija klasa za termove

Semantika logike prvog reda

Logika prvog reda ima značajno komlikovaniju semantiku u odnosu na iskaznu logiku. Na primer, rečenicu govornog jezika: *ako su **x** i **y** proizvoljni parni pozitivni celi brojevi tada je i zbir **x**+**y** takođe paran broj*, je moguće predstaviti formulom logike prvog reda na sledeći način:

$$\forall x\, \forall y\, ((paran(x) \wedge paran(y)) \Rightarrow paran(+(x, y)))$$

Postavlja se pitanje kako se možemo uveriti da je ovakva formula valjana? Direktan način je da zamenimo sve moguće kombinacije vrednosti za *x* i *y* i proverimo da je tvrđenje tačno za sve kombinacije. Iako neefikasan pristup, ovakav način rešavanja bi mogao da radi ukoliko *x* i *y* uzimaju vrednosti iz nekog konačnog skupa. S druge strane šta se dešava sa skupovima koji nisu konačni (kao što je na primer skup pozitivnih celih brojeva)? Dalje, kako znamo šta su značenja relacije *paran* i funkcije **+**? Signatura definiše sintaksno validne simbole, ne daje nam simbolima pridruženo značenje. Imajući to u vidu, prvo ćemo definisati aparat potreban za rezonovanje o semantici formula prvog reda. Dakle, potreban nam je skup iz koga biramo vrednosti promeljivih i konstanti i potrebna nam je interpretacija svih funkcijskih i relacijskih simbola (mapiranje između simbola i objekata koji zapravo vrše računanje).

*Def 6.7: **L**-strukturu odnosno model čine:*

- *Jezik tj. signatura **L***
- *Neprazan skup objekata **D** tj. domen*
- *Za svaki simbol konstante **c** njena interpretacija $\boldsymbol{c_d}$ iz **D***
- *Za svaki funkcijski simbol **f** arnosti **k** njena interpretacija tj. funkcija:*
  $\boldsymbol{f_d}$*:* $\boldsymbol{D^k \rightarrow D}$
- *Za svaki relacijski simbol **r** arnosti **k** njegova interpretacija tj. relacija:*
  $\boldsymbol{r_d \subseteq D^k}$

Na osnovu prethodne definicije može se ispravno pretpostaviti da nam je za određivanje vrednosti formule potrebna L-struktura. Osim nje, potrebna nam je i valuacija koja se u kontekstu logike prvog reda definiše kao funkcija koja dodeljuje vrednosti iz domena **D** promenljivama koje nisu vezane u formuli. Na primer, vrednost formule:

x+5<7 ili drugačije zapisano

<(+(x, 5), 7)

pri uobičajenoj interpretaciji simbola zavisi samo do vrednosti promenljive *x* koja se dohvata iz trenutne valuacije.

Da sumiramo, za određivanje vrednosti formule, pri čemu vrednost formule može biti tačno odnosno 1 ili netačno odnosno 0, biće nam potrebna L-struktura i valuacija. Na kraju možda nije loše naglasiti da je vrednost primene funkcije koja odgovara funkcijskom simbolu, tj. vrednost objekta klase *FunctionTerm*, vrednost iz **D**. S druge strane vrednost relacije dodeljene relacijskom simbolu, tj. vrednost objekta klase *Atom*, je 0 ili 1 u zavisnosti od toga da li su argumenti u relaciji ili ne.

Implementacija sintakse i semantike logike prvog reda

Signatura

Klasa *Signature* je prilično jednostavna. Sve što ova klasa treba da obezbedi je da mi možemo da proverimo da li je odgovarajući simbol deo jezika odnosno signature. Od vrsta simbola imamo samo

funkcijske i relacijske simbole, pa se tako implementacije ove klase svodi na tanak omotač oko dve mape (videti interfejs klase signature.h).

Domen, valuacija i L-struktura

Počnimo od domena. Razuman zahtev bi bio da klasa kojom se predstavi domen omogućava da se u domen dodaju vrednosti i da se za vrednost može proveriti da li pripada domenu. Dodatno, domen bi trebalo da bude proizvoljan neprazan skup, na primer skup niski, ili realnih brojeva ili celih brojeva. Skup vrednosti možemo predstaviti sa *std::set*, međutim postavlja se pitanje kako se izboriti sa opcijom da se i *std::string* i *unsigned* dodaju u *std::set* pri čemu su ove vrednosti različitog tipa, a C++ je strogo tipiziran programski jezik. Prirodno rešenje koje se nameće je da *Domain* bude šablonska klasa, međutim brzo se dolazi do problema jer bi onda virutalan metod *eval()*, koji implicitno prima domen i kojim se određuje vrednost formule ili terma, morao da bude šablonski, jer treba da primi kao argument *Domain<T>* za različite tipove *T*. Pošto C++ ne podržava virtualne šablonske metode ovo nije opcija. Slično, valuacija sadrži vrednosti iz domena, pa bi i valuacija morala da bude šablonska klasa. Takođe, L-struktura ima domen kao članicu klase pa isto važi i za ovaj entitet.

Standardna C++ biblioteka počevši od standarda C++17 podržava tip podataka *std::any* koji je u stanju da čuva vrednost bilo kojeg vrednosnog tipa (za standard C++14 u pitanju je *std::experimental::any*). Dakle, instanca klase *std::any* može čuvati u sebi i *std::string* i *unsigned* i *double* po potrebi. Tehnika koju *std::any* implementira naziva se *eliminisanje tipa* (eng. type erasure) i detalji vezani za ovu tehniku su van obima ovog kursa (zainteresovan čitalac može naći lepo objašnjenje na linku http://www.two-sdg.demon.co.uk/curbralan/papers/ValuedConversions.pdf).

Dakle sa *std::any* možemo da čuvamo proizvoljan tip, međutim tu nije kraj problema. Želimo da imamo isti izvorni kod klase za različite tipove, ali ne i da žrtvujemo bezbednost tipova. Na primer, želimo da izbegnemo:

```
Domain d;
d.insert("ovo je niska");
d.insert(3.14);
```

Postoji rešenje koje se bazira na kombinaciji šablonske klase i *std::any* tipa. Ideja je da uvedemo baznu klasu za sve domene, klasu *BaseDomain* u nastavku, koja u svom virtualnom interfejsu operiše sa *std::any* (u nastavku će se još koristiti i *AnyType* kao sinonim), a da izvedene klase budu šablonske i sadrže *std::set* za konkretne domene (npr. *std::set<std::string>* za niske, ili *std::set<double>* za realne brojeve). Svaka potklasa koja implementira virtualne metode zna tačno u koji tip da kastuje argumente koje prima kao *AnyType*. Na primer, klasa *DomainSpecialization<std::string>* bi sadržala članicu tipa *std::set<std::string>* i uvek kastovala argumente iz *AnyType* u *std::string*. Za prosleđivanje samog domena dovoljno je definisati *using Domain=std::shared_ptr<BaseDomain>* i koristiti konkretne domene kroz pokazivač na baznu klasu. Detalji implementacije se nalaze u domain.h, pri čemu bi čitalac pre ovoga trebao da konsultuje datoteku common.h u kojoj se nalazi makro *AnyTypeToType*. Ovaj makro služi za kastovanje objekta *AnyType* u konkretan tip, na primer *AnyTypeToType(int, x)* kastuje objekat *x* koji je tipa *AnyType* u celobrojnu vrednost.

Što se valuacije tiče, potrebno je napraviti mapu koja svakoj promenljivoj dodeljuje neku vrednost iz konkretnog domena. Prilikom same implementacije klase kojom predstavljamo valuaciju, klasa *Valuation*, možemo iskoristiti isti trik kao za klasu *BaseDomain* i konstruisati interfejs na osnovu tipa *AnyType*. Osim interfejsa, *Valuation* mora da sadrži i konkretan domen kao članicu klase jer je

potrebno proveriti da vrednosti koje se dodeljuju promenljivama zaista pripadaju željenom konkretnom domenu (detalji su dati u valuation.h).
Na kraju kada imamo formiran pojam domena i valuacije potrebno je zadati L-strukturu. Po definiciji, L-struktura sadrži signaturu i domen jer se nad njima definiše. Osim toga L-struktura predstavlja mapiranje između funkcijskih, odnosno relacijskih simbola i konkretnih funkcija, odnosno relacija. Dakle, direktno iz definicije zaključujemo da L-struktura sadrži članice domen, signaturu i dve mape za preslikavanje simbola u konkretne funkcije, odnosno relacije. S obzirom na to da treba da imamo mapu koja može da sadrži bilo koju funkciju pridruženu funkcijskom simbolu postoje bar dva načina da se ovo realizuje. Pre navođenja potencijalnih rešenja, naglasimo da konkretna funkcija treba da vrši izračunavanja na argumentima koji su vrednosti domena i čiji broj može biti nula u slučaju konstanti. Dakle, potrebno je nekako pozvati funkciju koja prima vektor argumenata iz domena. Kako bismo osigurali da se ovakva funkcija može kombinovati sa već predstavljenom arhitekturom ona mora da prima vektor *AnyType* objekata i da ih kastuje u željeni tip prilikom evaluacije. Jedno rešenje je da se koristi omotač oko *funkcijskih objekata* (eng. callable), to jest klasa *std::function*. Međutim, kako smo do sada koristili više objektno-orijentisani nego funkcionalni pristup, korišćen je drugi način. Mapu je moguće relizovati tako što se uvede bazna klasa za sve funkcije koja ima čisto virtualni metod *eval()*. U mapu se dodaju pokazivači na izvedene klase koje implementiraju ovaj metod. Na analogan način je rešena i mapa koja sadrži relacijske simbole i konkretne relacije (detalji interfejsa su dati u lstructure.h).

Hijerarhija termova

Čitalac bi već trebalo da ima iskustva sa razvijanjem sličnih hijerarhija klasa. Radi kompletnosti, potrebno je naglasiti da imamo baznu klasu *BaseTerm* i dve izvedene klase *VariableTerm* (promenljive su termovi), odnosno *FunctionTerm* (kada je arnost ovog terma nula u pitanju je konstanta, a inače funkcijski simbol u pravom smislu). Što se samog interfejsa koji termovi pružaju tiče, funkcionalnost termova je slična kao i za formule logike prvog reda. Dakle treba izračunati vrednost terma, ispisati ga, zameniti promenljivu termom u okviru tekućeg terma i slično. Razlika između termova i formula je u tome što termovi operišu u okviru interpretacije odnosno konkretne teorije. Na primer, u izrazu $\underline{x+y} < z$ podvučeni deo je term čija je vrednost u slučaju da je interpretacija teorija realnih brojeva takođe realan broj, dok je ceo izraz atomička formula gde je vrednost relacije < tačno ili netačno (ispunjeno ili ne). S druge strane i termovi su izrazi i kao takvi predstavljaju se drvetom izraza i imaju slične funkcionalnosti kao i formule (detalji interfejsa su dati u base_term.h, function_term.h i variable_term.h).

Hijerarhija formula

Sama hijerarhija formula se ne razlikuje previše u odnosu na isti problem vezan za iskaznu logiku i treba naglasiti da je logika odgovarajućih implementacionih izbora ista kao i ranije. Jedina razlika je u kvantifikatorima i tu se treba osvrnuti na implementaciju zamene promenljive termom i evaluaciju kvantifikovane formule. Za početak posmatrajmo formulu:

$$\exists y\,(y + 1 = x)$$

i zamenu promenljive termom *x->y*. Direktna zamena dovodi do formule:

$$\exists y\; y + 1 = y$$

što nije efekat koji želimo da postignemo zamenom. Posmatrajmo još jedan nezgodan primer, neka je data slična formula:

$$\forall x(x = x)$$

i zamena *x->t* gde je *t* term. Možda nije odmah očigledno, ali ako neko tvrđenje važi za sve vrednosti nekog domena, onda nije bitno kako zovemo tu promenljivu.
Iz prikazanih primera dolazimo do logike za implementaciju zamene promenljive termom u kvantifikovanoj formuli:

1. Ako je promenljiva koja se menja ista kao i kvantifikovana promenljiva ignorišemo zamenu jer ne bi smela da menja vrednost formule
2. Ako term kojim se promenljiva menja sadrži kvantifikovanu promenljivu:
    a. U kvantifikovanoj potformuli preimenovati kvantifikovanu promenljivu promenljivom koja se ne nalazi ni u termu ni u potformuli
    b. U izmenjenoj potformuli izvršiti originalnu zamenu

Dakle, za prvi prikazan primer prvo bismo radili preimenovanje *y->y'* pa onda *x->y* (*y'* je unikatna promenljiva) pri čemu se dobija:

$$\exists y'(y' + 1 = y)$$

Da bismo videli gde se računanje vrednosti kvantifikovane potformule razlikuje posmatrajmo vrednost formule:

$\forall x(x = x)$ za V1={'x'->5 ...} i V2={'x'->2 ...}

gde su *V1* i *V2* različite valuacije. Postavlja se pitanje da li vrednost prikazane formule zavisi od izabrane valuacije? Odgovor je da ne zavisi jer se jednakost proverava za svako *x* iz domena. Slično je i kada imamo egzistencijalni kvantifikator u formuli, on kaže da postoji bar jedna vrednost iz domena za koju je formula tačna i opet ne uzima u obzir konkretnu valuaciju. Ako gledamo sa stanovišta imlementacije u oba slučaja imamo prolazak kroz domen i dodeljivanje svih vrednosti domena kvantifikovanoj promenljivoj:

- U slučaju kvantifikatora ∀ ako naiđemo na dodelu gde potformula nije tačna onda ni naša kvantifikovana formula nije tačna
- U slučaju kvantifikatora ∃ ako naiđemo na dodelu gde je potformula tačna onda je i naša kvantifikovana formula tačna

U sekciji Zaglavlja se nalaze interfejsi svih ovih klasa, dok se u sekciji Izvorni kodovi nalaze njihove implementacije.

Zaglavlja

common.h

```
#ifndef COMMON_H
#define COMMON_H

#include <string>
#include <unordered_set>

#define UNUSED_ARG(x) ((void)x)

using Variable = std::string;
using FunctionSymbol = std::string;
using RelationSymbol = std::string;
using Arity = unsigned;
using VariablesSet = std::unordered_set<Variable>;
```

```
/* Reprezentacija bilo kog vrednosnog tipa */
#include <experimental/any>
using AnyType = std::experimental::any;
#define AnyTypeToType(type, x) std::experimental::any_cast<type>(x)

#endif // COMMON_H
```

signature.h

```
#ifndef SIGNATURE_H
#define SIGNATURE_H

#include "common.h"

#include <unordered_map>
#include <memory>

class Signature
{
public:
   using Sptr = std::shared_ptr<Signature>;
private:
 using Map = std::unordered_map<std::string, Arity>;
 public:
   void addFunctionSymbol(const FunctionSymbol &fsym, Arity ar);

   void addPredicateSymbol(const RelationSymbol &fsym, Arity ar);

   bool hasFunctionSymbol(const FunctionSymbol &fsym, const Arity &ar) const;

   bool hasPredicateSymbol(const RelationSymbol &psym, const Arity &ar) const;

private:
   Map m_functions;
   Map m_predicates;
};

#endif // SIGNATURE_H
```

domain.h

```
#ifndef DOMAIN_H
#define DOMAIN_H

#include "common.h"
```

```
#include <unordered_map>
#include <set>
#include <memory>
#include <vector>

class BaseDomain
{
public:
  virtual ~BaseDomain();

  virtual void insert(const AnyType &val) = 0;

  virtual bool hasValue(const AnyType &val) const = 0;

  virtual std::vector<AnyType> getValues() const = 0;
};

template <typename ValueType>
class DomainSpecialization : public BaseDomain
{
public:
  DomainSpecialization();

  virtual void insert(const AnyType &val);

  virtual bool hasValue(const AnyType &val) const;

  virtual std::vector<AnyType> getValues() const;
private:
  std::set<ValueType> m_values;
};

template <typename ValueType>
DomainSpecialization<ValueType>::DomainSpecialization()
  : BaseDomain ()
{
}

template <typename ValueType>
void DomainSpecialization<ValueType>::insert(const AnyType &val)
{
  m_values.insert(AnyTypeToType(ValueType, val));
}

template <typename ValueType>
bool DomainSpecialization<ValueType>::hasValue(const AnyType &val) const
```

```
{
  auto it = m_values.find(AnyTypeToType(ValueType, val));
  return it != m_values.cend();
}

template <typename ValueType>
std::vector<AnyType> DomainSpecialization<ValueType>::getValues() const
{
  std::vector<AnyType> allValues;
  allValues.reserve(m_values.size());
  for (const ValueType &val : m_values)
  {
     allValues.emplace_back(val);
  }
  return allValues;
}

using Domain = std::shared_ptr<BaseDomain>;

#endif // DOMAIN_H
```

valuation.h

```
#ifndef BASEVALUATION_H
#define BASEVALUATION_H

#include "common.h"
#include "domain.h"

#include <unordered_map>

class Valuation
{
public:
 using Map = std::unordered_map<Variable, AnyType>;
 public:
 Valuation(const Domain &d);
  void setValue(const Variable &v, const AnyType &val);
  const AnyType& getValue(const Variable &v) const;
private:
 Domain m_domain;
 Map m_map;
};

#endif // BASEVALUATION_H
```

lstructure.h

```
#ifndef LSTRUCTURE_H
#define LSTRUCTURE_H

#include "common.h"
#include "signature.h"
#include "domain.h"

#include <vector>
#include <memory>
#include <unordered_map>

class BaseFunction
{
public:
 BaseFunction(Arity arity);
  inline Arity getArity() const { return m_arity; }
  virtual AnyType eval(const std::vector<AnyType> &args) const = 0;
  virtual ~BaseFunction();
private:
 Arity m_arity;
};

class BaseRelation
{
public:
 BaseRelation(Arity arity);
  inline Arity getArity() const { return m_arity; }
  virtual bool eval(const std::vector<AnyType> &args) const = 0;
  virtual ~BaseRelation();
private:
 Arity m_arity;
};

using Function = std::shared_ptr<BaseFunction>;
using Relation = std::shared_ptr<BaseRelation>;

class LStructure
{
public:
 using FuncMap = std::unordered_map<FunctionSymbol, Function>;
 using RelMap = std::unordered_map<RelationSymbol, Relation>;
public:
   LStructure(Signature::Sptr signature, const Domain &d);
```

```
  inline const Domain& getDomain() const { return m_domain; }

  void addFunction(const FunctionSymbol &symbol, const Function &f);
  void addRelation(const RelationSymbol &symbol, const Relation &r);

  const Function& getFunction(const FunctionSymbol &symbol) const;
  const Relation& getRelation(const RelationSymbol &symbol) const;
private:
  Signature::Sptr m_signature;
  Domain m_domain;
  FuncMap m_fmap;
  RelMap m_rmap;
};

#endif // LSTRUCTURE_H
```

base_term.h

```
#ifndef BASETERM_H
#define BASETERM_H

#include "common.h"
#include "lstructure.h"
#include "valuation.h"

#include <memory>
#include <iostream>

class BaseTerm;

using Term = std::shared_ptr<BaseTerm>;

class BaseTerm : public std::enable_shared_from_this<BaseTerm>
{
public:
  /**
   * @brief BaseTerm konstruktor
   */
  BaseTerm();

  /**
   * @brief print - stampa term u C++ stream
   * @param out - izlazni stream
   * @return referencu na izmenjeni stream
   */
  virtual std::ostream& print(std::ostream &out) const = 0;
```

```
  /**
   * @brief equalTo - provera sintaksne jednakosti dva terma
   * @param oth - term sa kojim se tekuci objekat poredi
   * @return true ako su sintaksno jednaki, false inace
   */
  virtual bool equalTo(const Term &oth) const = 0;

  /**
   * @brief getVariables - dohvatanje svih promenljivih koje se javljaju u termu
   * @param vset - skup promenljivih na koji se nadovezuju promenljive ovog terma
   */
  virtual void getVariables(VariablesSet &vset) const = 0;

  /**
   * @brief hasVariable - provera da li term sadrzi promenljivu
   * @param v
   * @return
   */
  virtual bool hasVariable(const Variable &v) const = 0;

  /**
   * @brief substitute - zamena promenljive 'v' sa termom 't' u tekucem termu
   * @param v - promenljiva koja se menja
   * @param t - term kojim se promenljiva menja
   * @return izmenjeni term
   */
  virtual Term substitute(const Variable &v, const Term &t) const = 0;

  /**
   * @brief eval - racuna vrednost terma u tekucoj valuaciji i interpretaciji (rezultujuca vrednost je
vrednost iz domena)
   * @param structure - konkretna interpretacija
   * @param valuation - konkretna valuacija
   * @return vrednost terma
   */
  virtual AnyType eval(const LStructure &structure, const Valuation &valuation) const = 0;

  /**
   * @brief ~BaseTerm destruktor
   */
  virtual ~BaseTerm();
};

bool operator==(const Term &lhs, const Term &rhs);

#endif // BASETERM_H
```

function_term.h

```
#ifndef FUNCTIONTERM_H
#define FUNCTIONTERM_H

#include "base_term.h"
#include "signature.h"

#include <vector>

class FunctionTerm : public BaseTerm
{
public:
   FunctionTerm(Signature::Sptr signature, const FunctionSymbol &symbol, const
std::vector<Term> &terms = {});

   inline const FunctionSymbol& symbol() const { return m_symbol; }

   inline const std::vector<Term>& operands() const { return m_terms; }

   virtual std::ostream& print(std::ostream &out) const;

   virtual bool equalTo(const Term &oth) const;

   virtual void getVariables(VariablesSet &vset) const;

   virtual bool hasVariable(const Variable &v) const;

   virtual Term substitute(const Variable &v, const Term &t) const;

   virtual AnyType eval(const LStructure &structure, const Valuation &valuation) const;
private:
   /**
    * @brief m_signature je signatura kojoj funkcijski simbol pripada
    */
   Signature::Sptr m_signature;
   /**
    * @brief m_symbol je funkcijski simbol, na primer "+"
    */
   FunctionSymbol m_symbol;
   /**
    * @brief m_terms je vektor operanada, na primer funkcijski term arnosti 3 ima vektor
duzine 3
    */
   std::vector<Term> m_terms;
};

#endif // FUNCTIONTERM_H
```

variable_term.h

```
#ifndef VARIABLETERM_H
#define VARIABLETERM_H

#include "base_term.h"

class VariableTerm : public BaseTerm
{
public:
   VariableTerm(const Variable &var = {});

   inline const Variable& variable() const { return m_var; }

   virtual std::ostream& print(std::ostream &out) const;

   virtual bool equalTo(const Term &oth) const;

   virtual void getVariables(VariablesSet &vset) const;

   virtual bool hasVariable(const Variable &v) const;

   virtual Term substitute(const Variable &v, const Term &t) const;

   virtual AnyType eval(const LStructure &structure, const Valuation &valuation) const;
private:
   /**
    * @brief m_var je promenljiva koja odgovara termu, njena vrednost se cita iz
valuacije
    */
   Variable m_var;
};

#endif // VARIABLETERM_H
```

base_formula.h

```
#ifndef BASEFORMULA_H
#define BASEFORMULA_H

#include "common.h"
#include "base_term.h"

#include <iostream>
#include <memory>

class BaseFormula;

using Formula = std::shared_ptr<BaseFormula>;

class BaseFormula : public std::enable_shared_from_this<BaseFormula>
```

```
{
public:
   BaseFormula();

   virtual ~BaseFormula();

   virtual std::ostream& print(std::ostream & out) const = 0;

   virtual unsigned complexity() const = 0;
    virtual bool equalTo(const Formula & f) const;
    virtual void getVars(VariablesSet & vars, bool free = false) const = 0;
    bool containsVariable(const Variable & v, bool free = false) const;
    virtual bool eval(const LStructure &structure, const Valuation &valuation) const =
0;
    virtual Formula substitute(const Variable & v, const Term & t) const = 0;
};

std::ostream& operator<<(std::ostream &out, const Formula &f);

#endif // BASEFORMULA_H
```

atomic_formula.h

```
#ifndef ATOMICFORMULA_H
#define ATOMICFORMULA_H

#include "base_formula.h"
#include "common.h"

class AtomicFormula : public BaseFormula
{
public:
   AtomicFormula();

   virtual unsigned complexity() const;

   virtual void getVars(VariablesSet & vars, bool free) const;

   virtual Formula substitute(const Variable & v, const Term & t) const;
};

#endif // ATOMICFORMULA_H
```

constants.h

```
#ifndef CONSTANTS_H
#define CONSTANTS_H

#include "atomic_formula.h"

class True : public AtomicFormula
{
```

```
public:
   True();

   virtual std::ostream& print(std::ostream & out) const;

   virtual bool eval(const LStructure &structure, const Valuation &valuation) const;
};

class False : public AtomicFormula
{
public:
   False();

   virtual std::ostream& print(std::ostream & out) const;

   virtual bool eval(const LStructure &structure, const Valuation &valuation) const;
};

#endif // CONSTANTS_H
```

atom.h

```
#ifndef ATOM_H
#define ATOM_H

#include "atomic_formula.h"
#include "base_term.h"
#include "signature.h"

#include <vector>

class Atom : public AtomicFormula
{
public:
   Atom(Signature::Sptr signature, const RelationSymbol &symbol, const std::vector<Term>
&terms);

   inline const RelationSymbol& symbol() const { return m_symbol; }

   inline const std::vector<Term>& operands() const { return m_terms; }

   virtual std::ostream& print(std::ostream & out) const;
    virtual bool equalTo(const Formula & f) const;
    virtual void getVars(VariablesSet & vars, bool free = false) const;
    bool containsVariable(const Variable & v, bool free = false) const;
    virtual bool eval(const LStructure &structure, const Valuation &valuation) const;
    virtual Formula substitute(const Variable & v, const Term & t) const;

private:
   /**
    * @brief m_signature je signatura kojoj relacijski simbol 'm_symbol' mora da pripada
    */
```

```
    Signature::Sptr m_signature;
    /**
     * @brief m_symbol je simbol relacije, na primer '<' za relaciju 'manje od'
     */
    RelationSymbol m_symbol;
    /**
     * @brief m_terms je vektor termova koji su argumenti relacije, na primer
     * relacija '<' bi mogla da ima 2 argumenta koji se porede
     */
    std::vector<Term> m_terms;
};

#endif // ATOM_H
```

unary_connective.h

```
#ifndef UNARYCONNECTIVE_H
#define UNARYCONNECTIVE_H

#include "base_formula.h"

class UnaryConnective : public BaseFormula
{
public:
    UnaryConnective(const Formula &f);

    virtual unsigned complexity() const;
     virtual bool equalTo(const Formula & f) const;
     virtual void getVars(VariablesSet & vars, bool free = false) const;

protected:
    Formula m_op;
};

#endif // UNARYCONNECTIVE_H
```

not.h

```
#ifndef NOT_H
#define NOT_H

#include "unary_connective.h"

class Not : public UnaryConnective
{
public:
    Not(const Formula &f);

    virtual Formula substitute(const Variable & v, const Term & t) const;

    virtual std::ostream& print(std::ostream & out) const;
```

```
   virtual bool eval(const LStructure &structure, const Valuation &valuation) const;
};

#endif // NOT_H
```

binary_connective.h

```
#ifndef BINARYCONNECTIVE_H
#define BINARYCONNECTIVE_H

#include "base_formula.h"

class BinaryConnective : public BaseFormula
{
public:
   BinaryConnective(const Formula &op1, const Formula &op2);

   virtual unsigned complexity() const;
    virtual bool equalTo(const Formula & f) const;
    virtual void getVars(VariablesSet & vars, bool free = false) const;
 protected:
   template <typename Derived>
   Formula substituteImpl(const Variable & v, const Term & t) const;

   std::ostream& printImpl(std::ostream &out, const std::string &symbol) const;

protected:
   Formula m_op1;
   Formula m_op2;
};

template <typename Derived>
Formula BinaryConnective::substituteImpl(const Variable &v, const Term &t) const
{
 return std::make_shared<Derived>(m_op1->substitute(v, t), m_op2->substitute(v, t));
}

#endif // BINARYCONNECTIVE_H
```

and.h

```
#ifndef AND_H
#define AND_H

#include "binary_connective.h"

class And : public BinaryConnective
{
public:
   And(const Formula &op1, const Formula &op2);

   virtual std::ostream& print(std::ostream & out) const;
```

```
   virtual Formula substitute(const Variable & v, const Term & t) const;

   virtual bool eval(const LStructure &structure, const Valuation &valuation) const;
};

#endif // AND_H
```

or.h

```
#ifndef OR_H
#define OR_H

#include "binary_connective.h"

class Or : public BinaryConnective
{
public:
   Or(const Formula &op1, const Formula &op2);

   virtual std::ostream& print(std::ostream & out) const;

   virtual Formula substitute(const Variable & v, const Term & t) const;

   virtual bool eval(const LStructure &structure, const Valuation &valuation) const;
};

#endif // OR_H
```

imp.h

```
#ifndef IMP_H
#define IMP_H

#include "binary_connective.h"

class Imp : public BinaryConnective
{
public:
   Imp(const Formula &op1, const Formula &op2);

   virtual std::ostream& print(std::ostream & out) const;

   virtual Formula substitute(const Variable & v, const Term & t) const;

   virtual bool eval(const LStructure &structure, const Valuation &valuation) const;
};

#endif // IMP_H
```

iff.h

```
#ifndef IFF_H
```

```
#define IFF_H

#include "binary_connective.h"

class Iff : public BinaryConnective
{
public:
   Iff(const Formula &op1, const Formula &op2);

   virtual std::ostream& print(std::ostream & out) const;

   virtual Formula substitute(const Variable & v, const Term & t) const;

   virtual bool eval(const LStructure &structure, const Valuation &valuation) const;
};

#endif // IFF_H
```

quantifier.h

```
#ifndef QUANTIFIER_H
#define QUANTIFIER_H

#include <cstdint>

#include "base_formula.h"
#include "common.h"
#include "variable_term.h"

class Quantifier : public BaseFormula
{
private:
   static uint64_t s_UniqueCounter;

public:
   Quantifier(const Variable &var, const Formula &f);

   virtual unsigned complexity() const;

   virtual bool equalTo(const Formula & f) const;

   virtual void getVars(VariablesSet & vars, bool free = false) const;

   /**
    * @brief getUniqueVarName - vraca promenljivu koja se ne nalazi ni u formuli ni u termu
    *
    * @details Koristi se prilikom zamene promeljive termom, kada term kojim se menja sadrzi
    * promenljivu koja se javlja kvantifikovana (vezana) u formuli.
    * @param f - formula
    * @param t - term
    * @return jedinstvena promenljiva
    */
```

```
    static Variable getUniqueVarName(const Formula &f, const Term &t);

protected:
    std::ostream& printImpl(std::ostream & out, const std::string &symbol) const;

    template <typename Derived>
    Formula substituteImpl(const Variable & v, const Term & t) const;

protected:
    /**
     * @brief m_var je kvatifikovana promenljiva
     */
    Variable m_var;
    /**
     * @brief m_op je potformula na koju se kvantifikator primenjuje
     */
    Formula m_op;
};

template <typename Derived>
Formula Quantifier::substituteImpl(const Variable &v, const Term &t) const
{
    /* ako je promenljiva koja se menja jednaka kvantifikovanoj ignorisemo zamenu */
    if (v == m_var)
    {
        return std::const_pointer_cast<BaseFormula>(shared_from_this());
    }
    else
    {
        /* ako term sadrzi kvantifikovanu promenljivu moramo je preimenovati u potformuli */
        if (t->hasVariable(m_var))
        {
            Variable rename =
getUniqueVarName(std::const_pointer_cast<BaseFormula>(shared_from_this()), t);
            Formula renamedOp = m_op->substitute(m_var,
std::make_shared<VariableTerm>(rename));
            return std::make_shared<Derived>(rename, renamedOp->substitute(v, t));
        }
        else
        {
            /* u suprotnom zamena se izvrsava na uobicajen nacin */
            return std::make_shared<Derived>(m_var, m_op->substitute(v, t));
        }
    }
}

#endif // QUANTIFIER_H
```

exists.h

```
#ifndef EXISTS_H
#define EXISTS_H
```

```
#include "quantifier.h"

class Exists : public Quantifier
{
public:
   Exists(const Variable &var, const Formula &f);

   virtual Formula substitute(const Variable & v, const Term & t) const;

   virtual std::ostream& print(std::ostream & out) const;

   virtual bool eval(const LStructure &structure, const Valuation &valuation) const;
};

#endif // EXISTS_H
```

forall.h

```
#ifndef FORALL_H
#define FORALL_H

#include "quantifier.h"


class Forall : public Quantifier
{
public:
   Forall(const Variable &var, const Formula &f);

   virtual std::ostream& print(std::ostream & out) const;
    virtual Formula substitute(const Variable & v, const Term & t) const;

   virtual bool eval(const LStructure &structure, const Valuation &valuation) const;
};

#endif // FORALL_H
```

Izvorni kodovi

signature.cpp

```
#include "signature.h"

void Signature::addFunctionSymbol(const FunctionSymbol &fsym, Arity ar)
{
 m_functions[fsym] = ar;
}

void Signature::addPredicateSymbol(const RelationSymbol &fsym, Arity ar)
{
```

```
 m_predicates[fsym] = ar;
}

bool Signature::hasFunctionSymbol(const FunctionSymbol &fsym, const Arity &ar) const
{
 auto it = m_functions.find(fsym);
 if (it == m_functions.cend())
 {
   return false;
 }
 else
 {
   return it->second == ar;
 }
}

bool Signature::hasPredicateSymbol(const RelationSymbol &psym, const Arity &ar) const
{
 auto it = m_predicates.find(psym);
 if (it == m_predicates.cend())
 {
   return false;
 }
 else
 {
   return it->second == ar;
 }
}
```

domain.cpp

```
#include "domain.h"

BaseDomain::~BaseDomain()
{
}
```

valuation.cpp

```
#include "valuation.h"


Valuation::Valuation(const Domain &d)
   : m_domain(d)
{
}

void Valuation::setValue(const Variable &v, const AnyType &val)
{
   if (m_domain->hasValue(val))
   {
       m_map[v] = val;
```

```
    }
    else
    {
        throw std::runtime_error{"Value not in domain"};
    }
}

const AnyType &Valuation::getValue(const Variable &v) const
{
    return m_map.at(v);
}
```

lstructure.cpp

```
#include "lstructure.h"

BaseFunction::BaseFunction(Arity arity)
 : m_arity(arity)
{
}

BaseFunction::~BaseFunction()
{
}

BaseRelation::BaseRelation(Arity arity)
 : m_arity(arity)
{
}

BaseRelation::~BaseRelation()
{
}

LStructure::LStructure(Signature::Sptr signature, const Domain &d)
 : m_signature(signature), m_domain(d)
{
}

void LStructure::addFunction(const FunctionSymbol &symbol, const Function &f)
{
 if (m_signature->hasFunctionSymbol(symbol, f->getArity()))
 {
   m_fmap[symbol] = f;
 }
 else
 {
   throw std::runtime_error{"Function symbol or arity missmatch (not present in
Signature)"};
 }
}
```

```
void LStructure::addRelation(const RelationSymbol &symbol, const Relation &r)
{
 if (m_signature->hasPredicateSymbol(symbol, r->getArity()))
 {
   m_rmap[symbol] = r;
 }
 else
 {
   throw std::runtime_error{"Relation symbol or arity missmatch (not present in
Signature)"};
 }
}

const Function &LStructure::getFunction(const FunctionSymbol &symbol) const
{
 return m_fmap.at(symbol);
}

const Relation &LStructure::getRelation(const RelationSymbol &symbol) const
{
 return m_rmap.at(symbol);
}
```

base_term.cpp

```
#include "base_term.h"

BaseTerm::BaseTerm()
{
}

BaseTerm::~BaseTerm()
{
}

bool operator==(const Term &lhs, const Term &rhs)
{
   return lhs->equalTo(rhs);
}
```

variable_term.cpp

```
#include "variable_term.h"

VariableTerm::VariableTerm(const Variable &var)
   : m_var{var}
{

}

std::ostream &VariableTerm::print(std::ostream &out) const
{
```

```
    return out << m_var;
}

bool VariableTerm::equalTo(const Term &oth) const
{
    const VariableTerm *pOth = dynamic_cast<const VariableTerm*>(oth.get());
    if (pOth)
    {
        return pOth->variable() == variable();
    }
    else
    {
        return false;
    }
}

void VariableTerm::getVariables(VariablesSet &vset) const
{
    vset.insert(variable());
}

bool VariableTerm::hasVariable(const Variable &v) const
{
    return v == variable();
}

Term VariableTerm::substitute(const Variable &v, const Term &t) const
{
    if (v == m_var)
    {
        return t;
    }
    else
    {
        return std::const_pointer_cast<BaseTerm>(shared_from_this());
    }
}

AnyType VariableTerm::eval(const LStructure &structure, const Valuation &valuation)
const
{
    UNUSED_ARG(structure);
    return valuation.getValue(m_var);
}
```

function_term.cpp

```
#include "function_term.h"
#include "lstructure.h"

#include <algorithm>
#include <iterator>
```

```
#include <stdexcept>

FunctionTerm::FunctionTerm(Signature::Sptr signature, const FunctionSymbol &symbol,
const std::vector<Term> &terms)
 : BaseTerm (), m_signature{signature}, m_symbol{symbol}, m_terms{terms}
{
 if (!m_signature->hasFunctionSymbol(m_symbol, (unsigned)m_terms.size()))
 {
     throw std::runtime_error{"Syntax error in construction of function term"};
 }
}

std::ostream &FunctionTerm::print(std::ostream &out) const
{
 if (m_terms.empty())
 {
   return out << m_symbol;
 }
 else
 {
   out << m_symbol << "(";
   for (auto first = m_terms.cbegin(), last = m_terms.cend(); first + 1 != last;
++first)
   {
     (*first)->print(out);
     out << ", ";
   }
   m_terms.back()->print(out);
   return out << ")";
 }
}

bool FunctionTerm::equalTo(const Term &oth) const
{
 const FunctionTerm *pOth = dynamic_cast<const FunctionTerm*>(oth.get());
 if (pOth)
 {
   return symbol() == pOth->symbol() &&
       m_terms.size() == pOth->m_terms.size() &&
       std::equal(m_terms.cbegin(), m_terms.cend(), pOth->m_terms.cbegin());
 }
 else
 {
   return false;
 }
}

void FunctionTerm::getVariables(VariablesSet &vset) const
{
 for (const auto &term : m_terms)
 {
```

```
   term->getVariables(vset);
 }
}

bool FunctionTerm::hasVariable(const Variable &v) const
{
 for (const auto &term : m_terms)
 {
   if (term->hasVariable(v))
   {
     return true;
   }
 }
 return false;
}

Term FunctionTerm::substitute(const Variable &v, const Term &t) const
{
 std::vector<Term> terms;
 terms.reserve(m_terms.size());
 std::transform(m_terms.cbegin(), m_terms.cend(), std::back_inserter(terms), [&](const
Term &el) { return el->substitute(v, t); });
 return std::make_shared<FunctionTerm>(m_signature, m_symbol, terms);
}

AnyType FunctionTerm::eval(const LStructure &structure, const Valuation &valuation)
const
{
   Function f = structure.getFunction(m_symbol);
   std::vector<AnyType> termValues;
   termValues.reserve(m_terms.size());
   std::transform(m_terms.cbegin(), m_terms.cend(), std::back_inserter(termValues),
                  [&](const Term &t) {
      return t->eval(structure, valuation);
   });
   return f->eval(termValues);
}
```

base_formula.cpp

```
#include "base_formula.h"

#include <typeinfo>

BaseFormula::BaseFormula()
{}

BaseFormula::~BaseFormula()
{}

bool BaseFormula::equalTo(const Formula &f) const
{
```

```
   const BaseFormula *base = f.get();
   return typeid (*this) == typeid (*base);
}

bool BaseFormula::containsVariable(const Variable &v, bool free) const
{
 VariablesSet vset;
 getVars(vset, free);
 return vset.find(v) != vset.cend();
}

std::ostream& operator<<(std::ostream &out, const Formula &f)
{
   return f->print(out);
}
```

atomic_formula.cpp

```
#include "atomic_formula.h"

AtomicFormula::AtomicFormula()
 : BaseFormula ()
{
 }

unsigned AtomicFormula::complexity() const
{
   return 0;
}

void AtomicFormula::getVars(VariablesSet &vars, bool free) const
{
   UNUSED_ARG(vars);
   UNUSED_ARG(free);
}

Formula AtomicFormula::substitute(const Variable &v, const Term &t) const
{
   UNUSED_ARG(v);
   UNUSED_ARG(t);
   return std::const_pointer_cast<BaseFormula>(shared_from_this());
}
```

constants.cpp

```
#include "constants.h"

True::True()
 : AtomicFormula ()
{
 }
```

```
std::ostream &True::print(std::ostream &out) const
{
   return out << "T";
}

bool True::eval(const LStructure &structure, const Valuation &valuation) const
{
   UNUSED_ARG(structure);
   UNUSED_ARG(valuation);
   return true;
}

False::False()
 : AtomicFormula ()
{
 }

std::ostream &False::print(std::ostream &out) const
{
   return out << "F";
}

bool False::eval(const LStructure &structure, const Valuation &valuation) const
{
   UNUSED_ARG(structure);
   UNUSED_ARG(valuation);
   return false;
}
```

atom.cpp

```
#include "atom.h"

#include <algorithm>
#include <stdexcept>

Atom::Atom(Signature::Sptr signature, const RelationSymbol &symbol, const
std::vector<Term> &terms)
   : AtomicFormula (), m_signature(signature), m_symbol(symbol), m_terms(terms)
{
   if (!signature->hasPredicateSymbol(m_symbol, (unsigned)m_terms.size()))
   {
       throw std::runtime_error{"Syntax error in construction of Atom"};
   }
}

std::ostream& Atom::print(std::ostream &out) const
{
   if (m_terms.empty())
   {
     return out << m_symbol;
   }
```

```
    else
    {
      out << m_symbol << "(";
      for (auto first = m_terms.cbegin(), last = m_terms.cend(); first + 1 != last;
++first)
      {
        (*first)->print(out);
        out << ", ";
      }
      m_terms.back()->print(out);
      return out << ")";
    }
}

bool Atom::equalTo(const Formula &f) const
{
    const Atom *pF = dynamic_cast<const Atom*>(f.get());
    if (pF)
    {
        return symbol() == pF->symbol() &&
            m_terms.size() == pF->m_terms.size() &&
            std::equal(m_terms.cbegin(), m_terms.cend(), pF->m_terms.cbegin());
    }
    else
    {
        return false;
    }
}

void Atom::getVars(VariablesSet &vars, bool free) const
{
    UNUSED_ARG(free);
    for (const Term &t : m_terms)
    {
        t->getVariables(vars);
    }
}

bool Atom::eval(const LStructure &structure, const Valuation &valuation) const
{
    Relation r = structure.getRelation(m_symbol);
    std::vector<AnyType> termValues;
    termValues.reserve(m_terms.size());
    std::transform(m_terms.cbegin(), m_terms.cend(), std::back_inserter(termValues),
                   [&](const Term &t) {
       return t->eval(structure, valuation);
    });
    return r->eval(termValues);
}

Formula Atom::substitute(const Variable &v, const Term &t) const
```

```
{
   std::vector<Term> terms;
   terms.reserve(m_terms.size());
   std::transform(m_terms.cbegin(), m_terms.cend(), std::back_inserter(terms), [&](const
Term &el) { return el->substitute(v, t); });
   return std::make_shared<Atom>(m_signature, m_symbol, terms);
}
```

unary_connective.cpp

```
#include "unary_connective.h"

UnaryConnective::UnaryConnective(const Formula &f)
   : BaseFormula (), m_op(f)
{
 }

unsigned UnaryConnective::complexity() const
{
 return m_op->complexity() + 1;
}

bool UnaryConnective::equalTo(const Formula &f) const
{
 if (BaseFormula::equalTo(f))
 {
   return m_op->equalTo(static_cast<const UnaryConnective*>(f.get())->m_op);
 }
 return false;
}

void UnaryConnective::getVars(VariablesSet &vars, bool free) const
{
 m_op->getVars(vars, free);
}
```

not.cpp

```
#include "not.h"

Not::Not(const Formula &f)
   : UnaryConnective (f)
{
}

Formula Not::substitute(const Variable &v, const Term &t) const
{
   return std::make_shared<Not>(m_op->substitute(v, t));
}

std::ostream &Not::print(std::ostream &out) const
{
```

```
   out << "~(";
   m_op->print(out);
   return out << ")";
}

bool Not::eval(const LStructure &structure, const Valuation &valuation) const
{
   return !m_op->eval(structure, valuation);
}
```

binary_connective.cpp

```
#include "binary_connective.h"

BinaryConnective::BinaryConnective(const Formula &op1, const Formula &op2)
 : BaseFormula (), m_op1(op1), m_op2(op2)
{
}

unsigned BinaryConnective::complexity() const
{
 return 1 + m_op1->complexity() + m_op2->complexity();
}

bool BinaryConnective::equalTo(const Formula &f) const
{
 if (BaseFormula::equalTo(f))
 {
   return m_op1->equalTo(static_cast<const BinaryConnective*>(f.get())->m_op1) &&
       m_op2->equalTo(static_cast<const BinaryConnective*>(f.get())->m_op2);
 }
 return false;
}

void BinaryConnective::getVars(VariablesSet &vars, bool free) const
{
 m_op1->getVars(vars, free);
 m_op2->getVars(vars, free);
}

std::ostream &BinaryConnective::printImpl(std::ostream &out, const std::string &symbol)
const
{
 out << '(';
 m_op1->print(out);
 out << ' ' << symbol << ' ';
 m_op2->print(out);
 return out << ')';
}
```

and.cpp

```
#include "and.h"

And::And(const Formula &op1, const Formula &op2)
   : BinaryConnective (op1, op2)
{}

std::ostream &And::print(std::ostream &out) const
{
   return printImpl(out, "/\\");
}

Formula And::substitute(const Variable &v, const Term &t) const
{
   return substituteImpl<And>(v, t);
}

bool And::eval(const LStructure &structure, const Valuation &valuation) const
{
   return m_op1->eval(structure, valuation) && m_op2->eval(structure, valuation);
}
```

or.cpp

```
#include "or.h"

Or::Or(const Formula &op1, const Formula &op2)
   : BinaryConnective (op1, op2)
{

}

std::ostream &Or::print(std::ostream &out) const
{
   return printImpl(out, "\\/");
}

Formula Or::substitute(const Variable &v, const Term &t) const
{
   return substituteImpl<Or>(v, t);
}

bool Or::eval(const LStructure &structure, const Valuation &valuation) const
{
   return m_op1->eval(structure, valuation) || m_op2->eval(structure, valuation);
}
```

imp.cpp

```
#include "imp.h"
```

```
Imp::Imp(const Formula &op1, const Formula &op2)
   : BinaryConnective (op1, op2)
{

}

std::ostream &Imp::print(std::ostream &out) const
{
   return printImpl(out, "=>");
}

Formula Imp::substitute(const Variable &v, const Term &t) const
{
   return substituteImpl<Imp>(v, t);
}

bool Imp::eval(const LStructure &structure, const Valuation &valuation) const
{
   return !m_op1->eval(structure, valuation) || m_op2->eval(structure, valuation);
}
```

iff.cpp

```
#include "iff.h"

Iff::Iff(const Formula &op1, const Formula &op2)
   : BinaryConnective (op1, op2)
{
}

std::ostream &Iff::print(std::ostream &out) const
{
   return printImpl(out, "<=>");
}

Formula Iff::substitute(const Variable &v, const Term &t) const
{
   return substituteImpl<Iff>(v, t);
}

bool Iff::eval(const LStructure &structure, const Valuation &valuation) const
{
   return m_op1->eval(structure, valuation) == m_op2->eval(structure, valuation);
}
```

quantifier.cpp

```
#include "quantifier.h"

uint64_t Quantifier::s_UniqueCounter = 0U;

Quantifier::Quantifier(const Variable &var, const Formula &f)
```

```
 : BaseFormula (), m_var(var), m_op(f)
{
}

unsigned Quantifier::complexity() const
{
 return 1 + m_op->complexity();
}

bool Quantifier::equalTo(const Formula &f) const
{
 if (BaseFormula::equalTo(f))
 {
   return m_var == static_cast<const Quantifier*>(f.get())->m_var &&
       m_op->equalTo(static_cast<const Quantifier*>(f.get())->m_op);
 }
 return false;
}

void Quantifier::getVars(VariablesSet &vars, bool free) const
{
 VariablesSet tmp;
 m_op->getVars(tmp);
 if (free)
 {
   tmp.erase(m_var);
 }
 vars.insert(tmp.cbegin(), tmp.cend());
}

Variable Quantifier::getUniqueVarName(const Formula &f, const Term &t)
{
 Variable unique;
 do {
   unique = "uv" + std::to_string(s_UniqueCounter++);
 } while (f->containsVariable(unique) || t->hasVariable(unique));
 return unique;
}

std::ostream &Quantifier::printImpl(std::ostream &out, const std::string &symbol) const
{
 out << '(' << symbol << '.' << m_var << ")(";
 m_op->print(out);
 return out << ')';
}
```

exists.cpp

```
#include "exists.h"

Exists::Exists(const Variable &var, const Formula &f)
   : Quantifier (var, f)
```

```
{
}

Formula Exists::substitute(const Variable &v, const Term &t) const
{
   return substituteImpl<Exists>(v, t);
}

std::ostream &Exists::print(std::ostream &out) const
{
   return printImpl(out, "E");
}

bool Exists::eval(const LStructure &structure, const Valuation &valuation) const
{
   Valuation cpyValuation = valuation;
   std::vector<AnyType> domainValues = structure.getDomain()->getValues();
   for (const AnyType &val : domainValues)
   {
       cpyValuation.setValue(m_var, val);
       if (m_op->eval(structure, cpyValuation))
       {
           return true;
       }
   }

   return false;
}
```

forall.cpp

```
#include "forall.h"

Forall::Forall(const Variable &var, const Formula &f)
   : Quantifier (var, f)
{
}

std::ostream &Forall::print(std::ostream &out) const
{
   return printImpl(out, "V");
}

Formula Forall::substitute(const Variable &v, const Term &t) const
{
   return substituteImpl<Forall>(v, t);
}

bool Forall::eval(const LStructure &structure, const Valuation &valuation) const
{
   Valuation cpyValuation = valuation;
   std::vector<AnyType> domainValues = structure.getDomain()->getValues();
```

```
    for (const AnyType &val : domainValues)
    {
        cpyValuation.setValue(m_var, val);
        if (!m_op->eval(structure, cpyValuation))
        {
            return false;
        }
    }

    return true;
}
```

Čas 7 - logika prvog reda, pojednostavljivanje i normalne forme

Preduslovi: *časovi 3 i 6*;

Normalne forme u logici prvog reda

Što se normalnih formi u logici prvog reda tiče, prisustvo kvantifikatora komplikuje definisanje normalnih formi. Slično kao u iskaznoj logici, možemo primeniti iste identitete uprošćavanja na formule bez kvantifikatora (Pojednostavljivanje formula), dok za kvantifikovane formule važe sledeća pravila:

- $\forall x\, Formula$ postaje samo *Formula* ukoliko se promenljiva *x* ne nalazi slobodna u *Formula*
- $\exists x\, Formula$ postaje samo *Formula* ukoliko se promenljiva *x* ne nalazi slobodna u *Formula*

Rezon za ovo uprošćavanje je sledeći, ako se u potformuli *Formula* originalne kvantifikovane formule promenljiva *x* nalazi slobodna to znači da je baš kvantifikator originalne formule vezuje i ne smemo da ga ukinemo. U suprotnom, naš kvantifikator ne vezuje promenljivu jer se ne javlja slobodna i možemo ga izbaciti.

Nakon što pojednostavimo formulu, sledeći korak je transformacija u *NNF*. Osim pravila za nekvantifikovane formule, dodajemo još dva pravila za kvantifikovane formule:

- $\sim \forall x\, Formula$ se transformiše u $\exists x \sim Formula$
- $\sim \exists x\, Formula$ se transformiše u $\forall x \sim Formula$

Prirodno, našu implementaciju hijerarhije klasa za rad sa formulama logike prvog reda ćemo proširiti sa metodima *simplify()* i *nnf()* koji su već bili korišćeni za formule iskazne logike.

Podsetimo se sada da je osnovna poenta normalnih formi da pripremimo polaznu formulu za proceduru odlučivanja u datoj logici (u našem slučaju logika prvog reda). Neki od metoda odlučivanja su metod tabloa, odnosno rezolucija u logici prvog reda, pri čemu oba koriste postupak koji se naziva *skolemizacija*. Što se metoda tabloa tiče, skolemizacija se koristi da se eliminišu egzistencijalni kvantifikatori iz drveta tabloa, dok rezolucija kao ulaz očekuje formule u *Skolem normalnoj formi*. Mi ćemo se sada fokusirati baš na ovu normalnu formu.

Def 7.1: Formula logike prvog reda je u prenex normalnoj formi ukoliko je u NNF i oblika:

$$Q_1x_1Q_2x_2...Q_nx_n\ Formula$$

pri čemu su Q_i kvantifikatori, a potformula Formula ne sadrži kvantifikatore.

Treba napomenuti da za svaku formulu logike prvog reda postoji formula u prenex normalnoj formi koja joj je logički ekvivalentna.

Def 7.2: Formula logike prvog reda je u Skolem normalnoj formi ako je u prenex normalnoj formi i ako su svi kvantifikatori univerzalni (nema egzistencijalnih kvantifikatora).

Ponovo napomena, za svaku formulu logike prvog reda postoji formula u Skolem normalnoj formi koja je ekvizadovoljiva u odnosu na originalnu formulu. Dakle, nije logička ekvivalentnost, ali ekvizadovoljivost je dovoljno jak uslov za većinu primena.

Pažljiv čitalac bi već sada mogao da ima ideju kako će se otprilike od NNF-a stići do Skolem normalne forme. Dakle, potrebno je izvući kvantifikatore ka spolja, a zatim eliminisati egzistencijalne kvantifikatore na neki način (skolemizacijom). Što se izvlačenja kvantifikatora tiče, rekurzivno ćemo primenjivati sledeća pravila na formule:

- (Q_ix A) /\ B -> Q_ix(A /\ B) A /\ (Q_ix B) -> Q_ix(A /\ B)
- (Q_ix A) \/ B -> Q_ix(A \/ B) A \/ (Q_ix B) -> Q_ix(A \/ B)

pri čemu Q_i označava kvantifikator (pravila važe i za egzistencijalni i za univerzalni kvantifikator). Naravno, ukoliko se kvantifikovana promenljiva nalazi slobodna u potformuli na koju se kvantifikator ne odnosi, potrebno je preimenovati promenljivu *x* u kvantifikovanoj potformuli da ne bismo greškom vezali promenljivu koja je bila slobodna. Dodatno, u nekim slučajevima možemo uštedeti jedan kvantifikator, preciznije imamo skraćena pravila (skraćena u smislu da eliminišu jedan kvantifikator u rezultujućem izrazu):

- $\forall x$ A /\ $\forall x$ B -> $\forall x$ (A /\ B)
- $\exists x$ A \/ $\exists x$ B -> $\exists x$ (A \/ B)

Ovde treba naglasiti dve stvari, prva je da drugačija kombinacija logičkih veznika i kvantifikatora ne može da se iskoristi, a druga da je opet potrebno izvršiti preimenovanje ukoliko se kvantifikovane promenljive drugačije obeležavaju. Na primer, u prvoj potformuli promenljiva je *x*, a u drugoj je *y*, mi obe preimenujemo u *z* i kvantifikujemo *z* u rezultujućoj formuli, pri čemu se podrazumeva da se *z* ne javlja ni u jednoj od početnih potformula. U implementaciju dodajemo dva metoda, *pullQuantifiers()* koji iz direktnih potformula izvlači kvantifikatore i metod *prenex()* koji rekurzivno izvlači kvantifikatore i primenjuje prenex transformaciju na operande (za detalje pogledati implementaciju).

Poslednja stvar koja nam je ostala je skolemizacija. Kao ulaz pretpostavljamo formulu u prenex normalnoj formi. Ekvizadovoljiva formula se dobija tako što se eliminišu egzistencijalni kvantifikatori spolja ka unutra, pri čemu se umesto svake egzistencijalno kvatifikovane promenljive uvodi novi funkcijski simbol čija arnost odgovara broju univerzalnih kvantifikatora pre datog egzistencijalnog kvatifikatora pri čemu su argumenti ovog funkcijskog simbola baš te kvantifikovane promenljive. Da bi bilo malo jasnije, posmatrajmo primer:

$$\forall x \forall y \exists z\ x + y = z \text{ -> } \forall x \forall y\ x + y = fNovo(x, y)$$

promenljiva *z* je zamenjena novim funkcijskim simbolom *fNovo* koji kao argumente ima univerzalno kvantifikovane promenljive levo od egzistencijalnog kvantifikatora. Treba primetiti da je ovo prilično jednostavna transformacija, cela implementacija se nalazi u klasama za kvantifikatore, a metod koji dodajemo je *skolem()*.

Da rezimiramo, prevođenje u Skolem normalnu formu vršimo na sledeći način (*prenex()* u svojoj implementaciji poziva *pullQuantifiers()*):

Formula->simplify()->nnf()->prenex()->skolem()

Implementacija

Desna referenca

Pre nego što izlistamo implementacije navedenih metoda, pogledajmo potpis funkcije za skolemizaciju:
virtual Formula *skolem*(Signature::Sptr s, VariablesSet &&vars = {}) const;
U programskom jeziku C++ simboli *&&* predstavljaju desnu referencu, odnosno referencu na nešto što nije leva vrednost (na primer konstanta 5 nije leva vrednost, ne možemo napisati *int a; 5 = a;*). Jedna od upotreba desne reference je takozvano savršeno prosleđivanje (eng. perfect forwarding). Osnovna ideja je da se izbegne pozivanje konstruktora kopije prilikom prosleđivanja promenljive sa jednog stek okvira na drugi (pozivom funkcije formira se novi stek okvir). Umesto toga originalna promenljiva se *pomeri* na novi stek okvir. Desna referenca je deo C++11 standarda i šireg koncepta koji se naziva semantika pomeranja (eng. move semantics). Od objekta desnu referencu dobijamo pozivom *std::move* funkcije standardne C++ biblioteke.

Pojednostavljivanje

base_formula.cpp

```
Formula BaseFormula::simplify() const
{
   return std::const_pointer_cast<BaseFormula>(shared_from_this());
}
```

not.cpp

```
Formula Not::simplify() const
{
   Formula sop = m_op->simplify();
   if (BaseFormula::isOfType<True>(sop))
   {
       return std::make_shared<False>();
   }
   else if (BaseFormula::isOfType<False>(sop))
   {
       return std::make_shared<True>();
   }
   else
   {
       return std::make_shared<Not>(sop);
   }
}
```

and.cpp

```
Formula And::simplify() const
{
   Formula sop1 = m_op1->simplify();
```

```
    Formula sop2 = m_op2->simplify();
    if (BaseFormula::isOfType<True>(sop1))
    {
        return sop2;
    }
    else if (BaseFormula::isOfType<True>(sop2))
    {
        return sop1;
    }
    else if (BaseFormula::isOfType<False>(sop1) || BaseFormula::isOfType<False>(sop2))
    {
        return std::make_shared<False>();
    }
    else
    {
        return std::make_shared<And>(sop1, sop2);
    }
}
```

or.cpp

```
Formula Or::simplify() const
{
    Formula sop1 = m_op1->simplify();
    Formula sop2 = m_op2->simplify();
    if (BaseFormula::isOfType<True>(sop1) || BaseFormula::isOfType<True>(sop2))
    {
        return std::make_shared<True>();
    }
    else if (BaseFormula::isOfType<False>(sop1))
    {
        return sop2;
    }
    else if (BaseFormula::isOfType<False>(sop2))
    {
        return sop1;
    }
    else
    {
        return std::make_shared<Or>(sop1, sop2);
    }
}
```

imp.cpp

```
Formula Imp::simplify() const
{
    Formula sop1 = m_op1->simplify();
    Formula sop2 = m_op2->simplify();
    if (BaseFormula::isOfType<True>(sop1))
    {
        return sop2;
```

```
    }
    else if (BaseFormula::isOfType<False>(sop1) || BaseFormula::isOfType<True>(sop2))
    {
        return std::make_shared<True>();
    }
    else if (BaseFormula::isOfType<False>(sop2))
    {
        return std::make_shared<Not>(sop1)->simplify();
    }
    else
    {
        return std::make_shared<Imp>(sop1, sop2);
    }
}
```

iff.cpp

```
Formula Iff::simplify() const
{
    Formula sop1 = m_op1->simplify();
    Formula sop2 = m_op2->simplify();
    if (BaseFormula::isOfType<True>(sop1))
    {
        return sop2;
    }
    else if (BaseFormula::isOfType<True>(sop2))
    {
        return sop1;
    }
    else if (BaseFormula::isOfType<False>(sop1))
    {
        return std::make_shared<Not>(sop2)->simplify();
    }
    else if (BaseFormula::isOfType<False>(sop2))
    {
        return std::make_shared<Not>(sop2)->simplify();
    }
    else
    {
        return std::make_shared<Iff>(sop1, sop2);
    }
}
```

exists.cpp

```
Formula Exists::simplify() const
{
    Formula sop = m_op->simplify();
    if (sop->hasVariable(m_var, true))
    {
        return std::make_shared<Exists>(m_var, sop);
    }
```

```
    else
    {
        return sop;
    }
}
```

forall.cpp

```
Formula Forall::simplify() const
{
    Formula sop = m_op->simplify();
    if (sop->hasVariable(m_var, true))
    {
        return std::make_shared<Forall>(m_var, sop);
    }
    else
    {
        return sop;
    }
}
```

Transformacija u NNF

base_formula.cpp

```
Formula BaseFormula::nnf() const
{
    return std::const_pointer_cast<BaseFormula>(shared_from_this());
}
```

not.cpp

```
Formula Not::nnf() const
{
    Formula op = m_op->nnf();
    if (BaseFormula::isOfType<Not>(op))
    {
        return op;
    }

    const And *aop = BaseFormula::isOfType<And>(op);
    if (aop)
    {
        GET_OPERANDS_EXT(aop, aop1, aop2);
        return std::make_shared<Or>(std::make_shared<Not>(aop1)->nnf(),
std::make_shared<Not>(aop2)->nnf());
    }

    const Or *oop = BaseFormula::isOfType<Or>(op);
    if (oop)
    {
        GET_OPERANDS_EXT(oop, oop1, oop2);
```

```
        return std::make_shared<And>(std::make_shared<Not>(oop1)->nnf(),
std::make_shared<Not>(oop2)->nnf());
    }

    const Imp *imop = BaseFormula::isOfType<Imp>(op);
    if (imop)
    {
        GET_OPERANDS_EXT(imop, imop1, imop2);
        return std::make_shared<And>(imop1->nnf(), std::make_shared<Not>(imop2)->nnf());
    }

    const Iff *ifop = BaseFormula::isOfType<Iff>(op);
    if (ifop)
    {
        GET_OPERANDS_EXT(ifop, ifop1, ifop2);
        return std::make_shared<Or>(std::make_shared<And>(
                                        ifop1->nnf(),
std::make_shared<Not>(ifop2)->nnf()),
                                    std::make_shared<And>(
                                        std::make_shared<Not>(ifop1)->nnf(),
ifop2->nnf()));
    }

    const Forall *fop = BaseFormula::isOfType<Forall>(op);
    if (fop)
    {
        Formula fop1 = fop->operand();
        Variable var = fop->variable();
        return std::make_shared<Exists>(var, std::make_shared<Not>(fop1)->nnf());
    }

    const Exists *eop = BaseFormula::isOfType<Exists>(op);
    if (eop)
    {
        Formula eop1 = eop->operand();
        Variable var = eop->variable();
        return std::make_shared<Forall>(var, std::make_shared<Not>(eop1)->nnf());
    }

    return std::make_shared<Not>(op);
}
```

and.cpp

```
Formula And::nnf() const
{
    return std::make_shared<And>(m_op1->nnf(), m_op2->nnf());
}
```

or.cpp

```
Formula Or::nnf() const
```

```
{
   return std::make_shared<Or>(m_op1->nnf(), m_op2->nnf());
}
```

imp.cpp

```
Formula Imp::nnf() const
{
   return std::make_shared<Or>(std::make_shared<Not>(m_op1)->nnf(), m_op2->nnf());
}
```

iff.cpp

```
Formula Iff::nnf() const
{
   return std::make_shared<And>(
               std::make_shared<Or>(
                   std::make_shared<Not>(m_op1)->nnf(),
                   m_op2->nnf()),
               std::make_shared<Or>(
                   m_op1->nnf(),
                   std::make_shared<Not>(m_op2)->nnf()));
}
```

exists.cpp

```
Formula Exists::nnf() const
{
   return std::make_shared<Exists>(m_var, m_op->nnf());
}
```

forall.cpp

```
Formula Forall::nnf() const
{
   return std::make_shared<Forall>(m_var, m_op->nnf());
}
```

Transformacija u prenex normalnu formu

base_formula.cpp

```
Formula BaseFormula::pullQuantifiers() const
{
   return std::const_pointer_cast<BaseFormula>(shared_from_this());
}

Formula BaseFormula::prenex() const
{
   return std::const_pointer_cast<BaseFormula>(shared_from_this());
}
```

binary_connective.cpp

```
Formula BinaryConnective::pullQuantifiers() const
{
   /* Ako metod nije reimplementiran u potklasi bacamo izuzetak (npr. za Iff i Imp) */
   throw std::runtime_error{"Formula in NNF should not contain this connective!"};
}

Formula BinaryConnective::prenex() const
{
   /* Ako metod nije reimplementiran u potklasi bacamo izuzetak (npr. za Iff i Imp) */
   throw std::runtime_error{"Formula in NNF should not contain this connective
(prenex)!"};
}
```

and.cpp

```
Formula And::pullQuantifiers() const
{
   const Forall *fop1 = BaseFormula::isOfType<Forall>(m_op1);
   const Forall *fop2 = BaseFormula::isOfType<Forall>(m_op2);
   if (fop1 && fop2)
   {
       if (fop1->variable() == fop2->variable())
       {
           return std::make_shared<Forall>(fop1->variable(),
                                           std::make_shared<And>(
                                               fop1->operand(),
                                               fop2->operand())->pullQuantifiers());
       }
       else
       {
           Variable renamed = Quantifier::getUniqueVarName(fop1->operand(),
fop2->operand());
           return std::make_shared<Forall>(renamed, std::make_shared<And>(
                                               fop1->operand()
                                                   ->substitute(fop1->variable(),
std::make_shared<VariableTerm>(renamed)),
                                               fop2->operand()
                                                   ->substitute(fop2->variable(),
std::make_shared<VariableTerm>(renamed))
                                               )->pullQuantifiers());
       }
   }
   else if (fop1)
   {
       if (m_op2->hasVariable(fop1->variable(), true))
       {
           Variable renamed = Quantifier::getUniqueVarName(fop1->operand(), m_op2);
           return std::make_shared<Forall>(renamed, std::make_shared<And>(
```

```
fop1->operand()->substitute(fop1->variable(), std::make_shared<VariableTerm>(renamed)),
                                                m_op2)->pullQuantifiers());
        }
        else
        {
            return std::make_shared<Forall>(fop1->variable(), std::make_shared<And>(
                                                fop1->operand(),
m_op2)->pullQuantifiers());
        }
    }
    else if (fop2)
    {
        if (m_op1->hasVariable(fop2->variable(), true))
        {
            Variable renamed = Quantifier::getUniqueVarName(fop2->operand(), m_op1);
            return std::make_shared<Forall>(renamed,  std::make_shared<And>(
                                                m_op1,

fop2->operand()->substitute(fop2->variable(),

std::make_shared<VariableTerm>(renamed))
                                                )->pullQuantifiers());
        }
        else
        {
            return std::make_shared<Forall>(fop2->variable(),
std::make_shared<And>(m_op1, fop2->operand())->pullQuantifiers());
        }
    }

    const Exists *eop1 = BaseFormula::isOfType<Exists>(m_op1);
    const Exists *eop2 = BaseFormula::isOfType<Exists>(m_op2);
    if (eop1)
    {
        if (m_op2->hasVariable(eop1->variable(), true))
        {
            Variable renamed = Quantifier::getUniqueVarName(eop1->operand(), m_op2);
            return std::make_shared<Exists>(renamed, std::make_shared<And>(

eop1->operand()->substitute(eop1->variable(), std::make_shared<VariableTerm>(renamed)),
                                                m_op2)->pullQuantifiers());
        }
        else
        {
            return std::make_shared<Exists>(eop1->variable(), std::make_shared<And>(
                                                eop1->operand(),
m_op2)->pullQuantifiers());
        }
    }
    else if (eop2)
```

```
    {
        if (m_op1->hasVariable(eop2->variable(), true))
        {
            Variable renamed = Quantifier::getUniqueVarName(eop2->operand(), m_op1);
            return std::make_shared<Exists>(renamed,  std::make_shared<And>(
                                                m_op1,

eop2->operand()->substitute(eop2->variable(),

std::make_shared<VariableTerm>(renamed))
                                                )->pullQuantifiers());
        }
        else
        {
            return std::make_shared<Exists>(eop2->variable(),
std::make_shared<And>(m_op1, eop2->operand())->pullQuantifiers());
        }
    }

    return std::const_pointer_cast<BaseFormula>(shared_from_this());
}

Formula And::prenex() const
{
    return std::make_shared<And>(m_op1->prenex(), m_op2->prenex())->pullQuantifiers();
}
```

or.cpp

```
Formula Or::pullQuantifiers() const
{
    const Exists *eop1 = BaseFormula::isOfType<Exists>(m_op1);
    const Exists *eop2 = BaseFormula::isOfType<Exists>(m_op2);
    if (eop1 && eop2)
    {
        if (eop1->variable() == eop2->variable())
        {
            return std::make_shared<Exists>(eop1->variable(),
                                        std::make_shared<Or>(
                                            eop1->operand(),
                                            eop2->operand())->pullQuantifiers());
        }
        else
        {
            Variable renamed = Quantifier::getUniqueVarName(eop1->operand(),
eop2->operand());
            return std::make_shared<Exists>(renamed,
                                        std::make_shared<Or>(

eop1->operand()->substitute(eop1->variable(), std::make_shared<VariableTerm>(renamed)),

eop2->operand()->substitute(eop2->variable(), std::make_shared<VariableTerm>(renamed))
```

```
                                                )->pullQuantifiers());
        }
    }
    else if (eop1)
    {
        if (m_op2->hasVariable(eop1->variable(), true))
        {
            Variable renamed = Quantifier::getUniqueVarName(eop1->operand(), m_op2);
            return std::make_shared<Exists>(renamed, std::make_shared<Or>(

eop1->operand()->substitute(eop1->variable(), std::make_shared<VariableTerm>(renamed)),
                                                m_op2)->pullQuantifiers());
        }
        else
        {
            return std::make_shared<Exists>(eop1->variable(), std::make_shared<Or>(
                                                eop1->operand(),
m_op2)->pullQuantifiers());
        }
    }
    else if (eop2)
    {
        if (m_op1->hasVariable(eop2->variable(), true))
        {
            Variable renamed = Quantifier::getUniqueVarName(eop2->operand(), m_op1);
            return std::make_shared<Exists>(renamed,  std::make_shared<Or>(
                                                m_op1,

eop2->operand()->substitute(eop2->variable(),

std::make_shared<VariableTerm>(renamed))
                                                )->pullQuantifiers());
        }
        else
        {
            return std::make_shared<Exists>(eop2->variable(), std::make_shared<Or>(m_op1,
eop2->operand())->pullQuantifiers());
        }
    }

    const Forall *fop1 = BaseFormula::isOfType<Forall>(m_op1);
    const Forall *fop2 = BaseFormula::isOfType<Forall>(m_op2);
    if (fop1)
    {
        if (m_op2->hasVariable(fop1->variable(), true))
        {
            Variable renamed = Quantifier::getUniqueVarName(fop1->operand(), m_op2);
            return std::make_shared<Forall>(renamed, std::make_shared<Or>(

fop1->operand()->substitute(fop1->variable(), std::make_shared<VariableTerm>(renamed)),
                                                m_op2)->pullQuantifiers());
```

```
        }
        else
        {
            return std::make_shared<Forall>(fop1->variable(), std::make_shared<Or>(
                                                fop1->operand(),
m_op2)->pullQuantifiers());
        }
    }
    else if (fop2)
    {
        if (m_op1->hasVariable(fop2->variable(), true))
        {
            Variable renamed = Quantifier::getUniqueVarName(fop2->operand(), m_op1);
            return std::make_shared<Forall>(renamed,  std::make_shared<Or>(
                                                m_op1,

fop2->operand()->substitute(fop2->variable(),

std::make_shared<VariableTerm>(renamed))
                                                )->pullQuantifiers());
        }
        else
        {
            return std::make_shared<Forall>(fop2->variable(), std::make_shared<Or>(m_op1,
fop2->operand())->pullQuantifiers());
        }
    }

    return std::const_pointer_cast<BaseFormula>(shared_from_this());
}

Formula Or::prenex() const
{
    return std::make_shared<Or>(m_op1->prenex(), m_op2->prenex())->pullQuantifiers();
}
```

exists.cpp

```
Formula Exists::prenex() const
{
    return std::make_shared<Exists>(m_var, m_op->prenex());
}
```

forall.cpp

```
Formula Forall::prenex() const
{
    return std::make_shared<Forall>(m_var, m_op->prenex());
}
```

Transformacija u Skolem normalnu formu

base_formula.cpp

```
Formula BaseFormula::skolem(Signature::Sptr s, VariablesSet &&vars) const
{
   UNUSED_ARG(s);
   UNUSED_ARG(vars);
   return std::const_pointer_cast<BaseFormula>(shared_from_this());
}
```

exists.cpp

```
Formula Exists::skolem(Signature::Sptr s, VariablesSet &&vars) const
{
   FunctionSymbol replacementSymbol = s->getUniqueFunctionSymbol();
   s->addFunctionSymbol(replacementSymbol, vars.size());
   std::vector<Term> terms;
   terms.reserve(vars.size());
   std::transform(vars.cbegin(), vars.cend(), std::back_inserter(terms), [&](const
Variable &var)
   {
       return std::make_shared<VariableTerm>(var);
   });

   return m_op->substitute(m_var, std::make_shared<FunctionTerm>(s, replacementSymbol,
terms))->skolem(s, std::move(vars));
}
```

forall.cpp

```
Formula Forall::skolem(Signature::Sptr s, VariablesSet &&vars) const
{
   vars.insert(m_var);
   return std::make_shared<Forall>(m_var, m_op->skolem(s, std::move(vars)));
}
```

quantifier.h

```
template <typename T1, typename T2>
static Variable getUniqueVarName(const T1 &t1, const T2 &t2);

template <typename T1, typename T2>
Variable Quantifier::getUniqueVarName(const T1 &t1, const T2 &t2)
{
   Variable unique;
   do {
     unique = "uv" + std::to_string(s_UniqueCounter++);
   } while (t1->hasVariable(unique) || t2->hasVariable(unique));
   return unique;
}
```

Čas 8, 9 i 10 - uopštena zamena, unifikacija i rezolucija u logici prvog reda

Preduslovi: *čas 7, rezolucija u iskaznoj logici*;

Rezolucija u logici prvog reda je prilično kompleksan algoritam i kao takvom biće mu posvećeno dosta pažnje. Izlaganje u ovoj glavi je organizovano na sledeći način:

- Opis algoritama supstitucije, unifikacije i rezolucije
- Primeri gde je rezolucija ručno primenjena
- Implementacija binarne rezolucije sa grupisanjem u programskom jeziku C++
- Upotreba postojećeg dokazivača za logiku prvog reda - Vampire

S obzirom na to da se sve vreme bavimo istom tematikom ove celine nisu eksplicitno podeljene po časovima već je predviđeno odgovarajuće vreme da se sve kompletiraju. Ovakav redosled celina je izabran sa ciljem da čitaocu olakša savladavanje ove nastavne jedinice.

Motivacija

Jedna od često korišćenih procedura odlučivanja u logici prvog reda je rezolucija. Iako je na prvi pogled rezolucija jednostavan algoritam, bar u kontekstu iskaznih formula, u logici prvog reda situacija je malo komplikovanija. Za početak posmatrajmo sledeće formule:

- $\forall x \forall y \, (A(x) \Rightarrow B(x, y))$
- $\forall i \forall j \, (A(i) \Rightarrow B(i, j))$

Pažljiv čitalac bi mogao da primeti da su navedene dve formule logički ekvivalentne. Naime, ako nešto važi za svaku instancu onda nije bitno kako tu instancu obeležavamo. Dalje, posmatrajmo sledeću formulu koja se sastoji od dve klauze:

$$\forall x \forall y \, (P(x) \wedge \sim P(y))$$

Kada bismo ignorisali kvantifikatore i pokušali da primenimo iskaznu rezoluciju primenjivali bismo je na klauzama P(x) i ~P(y). S obzirom na to da jedan predikat zavisi od *x*, a drugi od *y* na prvi pogled čini se da ne možemo da izvedemo praznu klauzu. Međutim, podsetimo se da smo malopre došli do zaključka da za univerzalno kvantifikovane formule sam naziv vezanih promenljivih ne igra nikakvu ulogu. Ako malo izmenimo definiciju iskazne rezolucije, i preimenujemo *y* -> *x*, dobijamo klauze P(x) i ~P(x) i izvodimo praznu klauzu.

Neformalno, postupak odlučivanja možemo formirati tako što:

1. Transformišemo polaznu formulu primenom sledećih koraka:
    a. Uprošćavanje
    b. Transformacija u NNF
    c. Transformacija u prenex normalnu formu
    d. Skolemizacija
2. Obrišemo sve kvantifikatore (svi su univerzalni i nalaze se ispred potformule koja je bez kvantifikatora)
3. Posmatrajući formulu kao formulu iskazne logike transformišemo je u KNF
4. Primenimo rezoluciju sa opisanim preimenovanjem kada je potrebno

Što se koraka 4. tiče, to jest preimenovanja, ono se obavlja supstitucijom (zamenom). Ova zamena je uopštena u smislu da je skup potencijalno više parova <promenljiva, term> pri čemu se promenljiva menja termom. Problem pronalaska odgovarajućeg uopštenog preimenovanja, ukoliko ono postoji, se naziva unifikacija.

Uopštena zamena

Uopštena zamena se može gledati i kao mapa koja svakoj promenljivoj koja učestvuje u ovakvoj zameni dodeljuje term. Jedna od mogućnosti za predstavljanje je svakako:

```
using Substitution = std::map<Variable, Term>;
```

pri čemu u baznu klasu za termove dodajemo:

```
virtual Term substitute(const Substitution &s) const = 0;
```

dok u baznu klasu za formule dodajemo:

```
virtual Formula substitute(const Substitution &s) const = 0;
```

Logika implementacije je ponovo rekurzivna i moglo bi se reći prilično jednostavna. U klasi *VariableTerm* prolazimo kroz sve parove iz supstitucije i ako naiđemo da se simbol promenljive iz nekog para supstitucije poklapa sa simbolom promenljive našeg objekta, tada vratimo kao povratnu vrednost term koji je pridružen promenljivoj. U klasu *FunctionTerm* rekurzivno izvršimo supstituciju za sve operande, pri čemu to važi i za sve formule osim kvantifikovanih formula gde je potrebno posvetiti posebnu pažnju zbog kvantifikovane promenljive.

Slično kao kod zamene promenljive termom, prolazimo kroz sve parove iz supstitucije i ignorišemo one kod kojih je simbole promenljive koja se menja jednak simbolu kvantifikovane promenljive u formuli. Ako ne postoji baš nijedan par koji nije takav, prekidamo supstituciju jer nemamo šta da zamenimo. Za sve parove koji nisu ignorisani, proveravamo da li bar jedan od tih termova sadrži kvantifikovanu promenljivu. Ako je odgovor da, preimenujemo promenljivu u kvantifikovanoj formuli, dok je u suprotnom bezbedno da nastavimo bez preimenovanja. Svakako na kraju pozivamo supstituciju nad potformulom koja je operand i skupom profiltriranih parova (onih parova koji ne sadrže kvantifikovanu promenljivu).

Unifikacija i najopštiji unifikator

Neformalno, ako su data dva izraza i supstitucija, kažemo da je supstitucija unifikator za ova dva izraza ukoliko se njenom primenom na oba izraza dobija isti izraz. Za same izraze kažemo da su unifikabilni. Bitna napomena je da za proizvoljne izraze ne postoji uvek supstitucija koja je unifikator, kao i da može postojati više različitih supstitucija koje jesu unifikatori. Pogledajmo par primera:

- P(x, g(y, x)), P(a, g(b, a)), izrazi su unifikabilni: [x -> a, y -> b]
- P(x, g(y, x)), P(a, f(b, a)), izrazi nisu unifikabilni jer se funkcijski simboli *f* i *g* razlikuju
- g(x, x), g(y, f(y)), izrazi nisu unifikabilni
- g(x, z), g(y, f(y)), izrazi su unifikabilni, imamo više unifikatora: [x -> y, z -> f(y)], [x -> a, y -> a, z -> f(a)]

Nas će u nastavku zanimati najopštiji unifikator. Ponovo neformalno, najopštiji unifikator je takav da se svaki drugi unifikator može dobiti od njega primenom neke supstitucije. Za poslednji primer, kada se na prvi unifikator primeni supstitucija [y -> a] dobije se drugi unifikator.

Pre navođenja samog algoritma za određivanje najopštijeg unifikatora treba naglasiti da on operiše nad parovima termova. To znači da bismo u nekom trenutku izvršavanja algoritma mogli da imamo preslikavanje koje izgleda nekako ovako {f(x, y) -> f(a, b), z -> g(c)}. Na samom kraju ovi parovi bi

mogli da proizvedu supstituciju [x -> a, y -> b, z -> g(c)]. Algoritam za pronalaženje najopštijeg unifikatora može se sumirati sledećim koracima:

1. Iz skupa parova termova (u nastavku samo skupa parova) izbaciti duplikate (eng. factorization)
2. Iz skupa parova izbaciti parove oblika $<t_i, t_j>$ gde je $t_i = t_j$ (eng. tautology)
3. Transformisati sve parove oblika <t, v> u <v, t> pri čemu je v promenljiva, a t term koji nije promenljiva (eng. orientation)
4. Ako imamo par $<t_i, t_j>$ pri čemu ni jedan od termova nije promenljiva:
    a. Ako termovi imaju iste funkcijske simbole i istu arnost dodati parove operanada u skup parova i izbaciti par $<t_i, t_j>$ (eng. decomposition)
    b. Inače, vratiti da ne postoji unifikator (eng. collision)
5. Ako imamo par <v, t> gde je v promenljiva i t term koji sadrži v vratiti da ne postoji unifikator (eng. cycle)
6. Ako imamo par <v, t>, gde je v promenljiva i t term koji ne sadrži v, primeniti supstituciju [v -> t] na sve parove i izbaciti par <v, t> iz skupa parova (eng. application)
7. Ako nije moguće primeniti nijedno pravilo vratiti tekući skup parova koji je najopštiji unifikator

Na kraju svakog od koraka dat je i naziv koji se za ove korake koristi u engleskoj literaturi.
Parove termova jednostavno predstavljamo vektorom parova jer je teško definisati uređenje nad deljenim pokazivačima pa ne možemo direktno da koristimo mapu (nije baš teško ali unosi određenu kompleksnost):

```
using TermPairs = std::vector<std::pair<Term, Term>>;
```

Za svaki od koraka napisaćemo funkciju koja ga implementira, kao i još po neku funkciju od kojih će svaka predstavljati omotač za pozivanje implementacije unifikacije (npr. po završetku postupka potrebno je konvertovati *TermPairs* u *Substitution*).

Rezolucija

Na početku ovog poglavlja rezolucija je data u potpunosti neformalno sa ciljem da formira neku intuiciju kod čitaoca i opravda smer izlaganja materijala u kontekstu zašto se uopšte bavimo supstitucijom i unifikacijom. S druge strane, čitalac sa većim predznanjem o materiji bi mogao da zna da problem zadovoljivosti u logici prvog reda nije odlučiv i samim tim mogao bi da očekuje određeni nivo komplikacija u formiranju algoritma za rezonovanje. U nastavku teksta biće izneta neka tvrđenja vezana za metod rezolucije koja neće biti dokazana u ovim materijalima jer je to van opsega izlaganja. Ohrabrujemo zainteresovanog čitaoca da za dokaze ovih tvrđenja i njihovo formalnije izlaganje konsultuje ove izvrsne materijale.
Fokusiraćemo se na problem *binarne rezolucije*. To znači da u svakom koraku za dve klauze eliminišemo jedan literal. Formalnije, primenjujemo pravilo:

$$\text{Klauza} \vee A \qquad \text{Klauza'} \vee \neg A' \rightarrow (\text{Klauza} \vee \text{Klauza'})\sigma$$

pri čemu je σ najopštiji unifikator za literale A i A'. Metod binarne rezolucije se sastoji od uzastopnog primenjivanja navedenog pravila i zaustavlja se ako ne postoje klauze na koje se pravilo može primeniti tako da se skup klauza promeni (u ovom slučaju formula je zadovoljiva) ili ako izvedemo praznu klauzu (formula je nezadovoljiva). Za metod rezolucije važe sledeće osobine:

- Metod rezolucije je saglasan, preciznije ako je metodom rezolucije dobijena prazna klauza polazni skup klauza je sigurno nezadovoljiv (nije moguće izvesti praznu klauzu iz zadovoljivog skupa klauza)

- Metod rezolucije je potpun za pobijanje, preciznije iz svakog nezadovoljivog sklupa klauza **moguće** je izvesti praznu klauzu
- Metod rezolucije je procedura za poluodlučivanje, za negaciju svake valjane formule imamo garanciju da se može izvesti prazna klauza

Dakle, metod rezolcije **može** izvesti praznu klauzu za svaku formulu koja nije zadovoljiva, ali se ne mora nikad zaustaviti za formulu koja jeste zadovoljiva. Dodatno, dva puta je naglašeno **može** jer iz skupa klauza možemo na više načina birati klauze za primenu pravila. Binarna rezolucija sama po sebi nema svojstvo kompletnosti tj. može da se desi da se njome ne izvede prazna klauza (dopunićemo je kasnije da to sprečimo). Opšta rezolucija takođe ne garantuje kompletnost, već zavisi od načina kako biramo klauze za rezolviranje. Pravilo opšte rezolucije je:

$$\text{Klauza} \vee A_1 \vee \ldots \vee A_m \quad \text{Klauza'} \vee \neg A_1' \vee \ldots \vee \neg A_n' \rightarrow (\text{Klauza} \vee \text{Klauza'})\,\sigma$$

pri čemu je σ najopštiji unifikator za navede literale A_i i A_j'. Jedna jednostavna strategija koja garantuje kompletnost je da primenimo pravilo opšte rezolucije na svaki par tekućeg skupa klauza. Ova strategija je kompletna, ali neefikasna i postoje dosta efikasnije strategije.

Vratimo se na našu binarnu rezoluciju koju ćemo implementirati. Da bismo napravili da bude kompletna dodajemo pravilo grupisanja (eng. factoring):

$$\text{Klauza} \vee A_1 \vee \ldots \vee A_n \rightarrow (\text{Klauza} \vee A_1)\,\sigma$$

pri čemu je σ najopštiji unifikator za literale A_i. Detalji vezani za dodavanje ovog pravila i kompletnost se mogu naći na primer u knjizi "*Artificial Intelligence: A Modern Approach 3rd edition"* (autori su Stjuart Rasel i Piter Norvig, strana je 350).

Da rezimiramo naš algoritam rezolucije se svodi na:

1. U beskonačnoj petlji
    a. Primeni grupisanje na sve klauze i zapamti da li je izmenjen skup klauza
    b. Primenjuj binarnu rezoluciju na sve parove klauza dokle god je to moguće i zapamti da li je promenjen skup klauza
    c. Ako je izvedena prazna klauza vrati da je formula nezadovoljiva
    d. Ako skup klauza nije promenjen u ovoj iteraciji (koraci a. i b. nisu promenili skup klauza), vrati da je formula zadovoljiva

Što se implementacije tiče, predstavljeni algoritam će biti razbijen u više funkcija, a konkretni detalji stoje u komentarima u samoj implementaciji. Napomena i mesta gde treba biti pažljiv prilikom implementacije ima dovoljno da njihovo navođenje van konteksta programskog koda ne bi bilo informativno. Jednostavnije je referisati na konkretne probleme direktno na licu mesta.

Primeri

Ova sekcija sadrži ručno rešavane primere rezolucije. Ideja je da se čitalac prolaskom kroz korake rešavanja još bliže upozna sa algoritmom rezolucije i da razvije odgovarajuću intuiciju koja će mu pomoći da lakše savlada implementaciju i njene detalje.

Primer 1

Metodom rezolucije dokazati (H /\ K) => L za potformule:

H = (Ax)(Ay)(p(x,y)=>p(y,x))
K = (Ax)(Ay)(Az)((p(x,y)/\p(y,z)) => p(x,z))
L = (Ax)((Ey)p(x,y) => p(x,x))

Rešenje:
Dokazujemo negaciju originalne implikacije tj. da je formula:
H /\ K /\ ~L
nezadovoljiva. Kada raspišemo taj izraz dobijamo:
(Ax)(Ay)(p(x,y)=>p(y,x))
(Ax)(Ay)(Az)((p(x,y)/\p(y,z)) => p(x,z))
(Ex)((Ey)p(x,y) /\ ~p(x,x))
Međukorak koji daje zadnji red raspisanog izraza je: (Ex)(~(~(Ey)p(x,y) \/ p(x,x)).
Naredna stavka je prevođenje u NNF čime se dobija (razložili smo implikacije):
(Ax)(Ay)(~p(x,y) \/ p(y,x)) /\
(Ax)(Ay)(Az)(~p(x,y) \/ ~p(y,z) \/ p(x,z)) /\
(Ex)((Ey)p(x,y) /\ ~p(x,x))
Nakon NNF transformacije potrebno je izvući kvantifikatore ispred formule tj. primeniti transformaciju u prenex normalnu formu u dva koraka (izvršili smo i dva preimenovanja da bismo izbegli konflikte; označena su masnim slovima):
(Ex)(E**v**)((Ax)(Ay)(~p(x,y) \/ p(y,x)) /\
(Ax)(Ay)(Az)(~p(x,y) \/ ~p(y,z) \/ p(x,z)) /\
(p(x,**v**) /\ ~p(x,x)))

(Ex)(Ev)(A**u**)(Ay)(Az)
(~p(**u**,y) \/ p(y,**u**)) /\
(~p(**u**,y)\/~p(y,z) \/ p(**u**,z)) /\
(p(x,v) /\ ~p(x,x))
U prvom koraku smo izvukli egzistencijalne kvantifikatore, a zatim univerzalne. Ovaj redosled je uvek poželjan jer pojednostavljuje naredni korak - skolemizaciju. Rezultat skolemizacije u ovom slučaju je zamena promenljivih *x* i *v* konstantama *a* i *b* (funkcijski simboli arnosti nula):
(Au)(Ay)(Az)
(~p(u,y) \/ p(y,u)) /\
(~p(u,y)\/~p(y,z) \/ p(u,z)) /\
(p(a,b) /\ ~p(a,a))
Direktnom eliminacijom kvantifikatora nad našom formulom dobija se formula koja je u KNF-u. Da to nije slučaj morali bismo da svedemo formulu na KNF. Radi lakšeg praćenja primene rezolucije indeksiraćemo klauze:
1) ~p(u,y), p(y,u)
2) ~p(u,y), ~p(y,z), p(u,z)
3) p(a,b)
4) ~p(a,a)
Prilikom primene rezolucije nad ovim klauzama zapisivaćemo koja rezolventa se dobila, iz kojih klauza i koja je supstitucija primenjena da bi se literali unifikovali. Koraci za tekuću formulu su sledeći:
5) p(b, a) (1, 3) u->a, y -> b
6) ~p(b, z), p(a, z) (2, 3) u->a, y->b
7) p(a,a) (5, 6) z->a
8) [] (4, 7)
Na primer za korak 5) važi da je rezolventa *p(b, a)*, da je dobijena iz klauza 1) i 3) i da je primenjena supstitucija [u->a, y->b]. S obzirom na to da smo uspešno izveli praznu klauzu polazno tvrđenje je

dokazano. U narednim rešenjima će biti preskočeni neki koraci i biće smanjen nivo detalja prilikom izlaganja rešenja. Naravno, iste konvencije koje smo ovde koristili će takođe biti primenjivane u tim rešenjima.

Primer 2

Dokazati (P /\ Q) => R:
P = (Ax)((s(x) /\ t(x)) => r(x)) => (Ex)(s(x) /\ ~t(x))
Q = (Ax)(s(x) => t(x)) \/ (Ax)(s(x) => r(x))
R = (Ax)((s(x) /\ r(x)) => t(x)) => (Ex)(s(x)/\t(x)/\~r(x))
Rešenje:
Dokazujemo da je P /\ Q /\ ~R nezadovoljiva:
(Ax)((s(x) /\ t(x)) => r(x)) => (Ex)(s(x) /\ ~t(x))
(Ax)(s(x) => t(x)) \/ (Ax)(s(x) => r(x))
(Ax)((s(x) /\ r(x)) => t(x)) /\ (Ax)(~s(x)\/~t(x)\/r(x))
NNF:
(Ex)(s(x) /\ t(x) /\ ~r(x)) \/ (Ex)(s(x) /\ ~t(x))
(Ax)(~s(x) \/ t(x)) \/ (Ax)(~s(x) \/ r(x))
(Ax)(~s(x) \/ ~r(x) \/ t(x)) /\ (Ax)(~s(x)\/~t(x)\/r(x))
PRENEX (primetiti da je iskorišćeno slaganje egzistencijalnog kvatifikatora sa disjunkcijom pa dobijamo jedan egzistencijalni kvantifikator umesto dva):
(Ex)(Av)(Au)
((s(x) /\ t(x) /\ ~r(x)) \/ (s(x) /\ ~t(x)))
((~s(v) \/ t(v)) \/ (~s(u) \/ r(u)))
((~s(v) \/ ~r(v) \/ t(v)) /\ (~s(v)\/~t(v)\/r(v)))
SKOLEMIZACIJA:
(Av)(Au)
((s(c) /\ t(c) /\ ~r(c)) \/ (s(c) /\ ~t(c)))
((~s(v) \/ t(v)) \/ (~s(u) \/ r(u)))
((~s(v) \/ ~r(v) \/ t(v)) /\ (~s(v)\/~t(v)\/r(v)))
KLAUZE:
1) s(c)
2) s(c), ~t(c)
3) t(c), s(c)
4) ~r(c), s(c)
5) ~r(c), ~t(c)
6) ~s(v), t(v), ~s(u), r(u)
7) ~s(v),~r(v),t(v)
8) ~s(v), ~t(v), r(v)
REZOLUCIJA:
9) t(c), r(c) (1, 6) v -> c, u->c
10) ~r(c),t(c) (1, 7) v-> c
11) ~t(c), r(c) (1,8) v->c
12) ~t(c) (11, 5)
13) t(c) (9,10)
14) [] (12, 13)

Primer 3

Dokazati da je sledeca formula valjana:
((Ax)(p(x) => q(x)) /\
(Ax)(q(x) => s(x)) /\
(Ax)(r(x) => s(x)) /\
(Ax)(p(x) \/ r(x))) => (Ax)s(x)
Rešenje:
Ponovo valjanost formule pokazujemo tako što je prvo negiramo, pa onda izvedemo praznu klauzu.
Negacija polazne formule uz rastavljanje implikacije je:
((Ax)(p(x) => q(x)) /\
(Ax)(q(x) => s(x)) /\
(Ax)(r(x) => s(x)) /\
(Ax)(p(x) \/ r(x))) /\ ~(Ax)s(x)
NNF:
((Ax)(~p(x) \/ q(x)) /\
(Ax)(~q(x) \/ s(x)) /\
(Ax)(~r(x) \/ s(x)) /\
(Ax)(p(x) \/ r(x))) /\ (Ex)~s(x)
PRENEX:
(Ex)(Ay)
(~p(y) \/ q(y)) /\
(~q(y) \/ s(y)) /\
(~r(y) \/ s(y)) /\
(p(y) \/ r(y))) /\ ~s(x)
SKOLEMIZACIJA:
(Ay)
(~p(y) \/ q(y)) /\
(~q(y) \/ s(y)) /\
(~r(y) \/ s(y)) /\
(p(y) \/ r(y))) /\ ~s(c)
KLAUZE:
1) (~p(y) \/ q(y))
2) (~q(y) \/ s(y))
3) (~r(y) \/ s(y))
4) (p(y) \/ r(y)))
5) ~s(c)
REZOLUCIJA:
6) ~q(c) (2, 5) y -> c
7) ~r(c) (3, 5) y -> c
8) ~p(c) (1, 6) y -> c
9) p(c) (4, 7) y -> c
10) [] (8,9)

Rezolucija u prisustvu jednakosti

Relacija jednakosti označena simbolom **=** je pristuna u velikom broju teorija čije se teoreme dokazuju dokazivačima za logiku prvog reda. Malo strožija definicija jednakosti će biti razmatrana kada budemo obrađivali rezonovanje u normalnim jednakosnim logikama (na primer koje su aksiome jednakosti i slično), za sada smatramo da je značenje simbola **=** intuitivno i jasno.
Kada imamo klauzu koja sadrži literal oblika $t_i = t_j$ postoji dodatno pravilo kojim se standardni algoritam rezolucije često proširuje. U pitanju je takozvana paramodulacija. Neka su C_1 i C_2 klauze i s, t i t' termi, tada važi:

$$C_1 \vee t = s \qquad C_2[t'] \rightarrow (C_1 \vee C_2[s])\,\sigma$$

pri čemu je σ najopštiji unifikator za t i t'. Da bi opšte pravilo bilo malo jasnije pogledajmo jednostavniju verziju:

$$t = s \quad C[t] \rightarrow C[s]$$

Dakle, rezon je sledeći, ukoliko su *t* i *s* jednaki i *t* se pojavljuje u klauzi *C* onda možemo proizvesti novu klauzu koja je identična klauzi *C* osim što su sva pojavljivanja terma *t* zamenjena sa termom *s* što intuitivno opravdamo time da su oni jednaki. Prvo uopštavanje ovog jednostavnog pravila koje možemo uvesti je da *C* ne mora sadržati baš term *t* nego recimo može sadržati term *t'* za koji važi da je unifikabilan sa *t*. I na kraju ako dodamo da leva klauza ne mora biti samo jedna jednakost već klauza može sadržati i druge literale dobićemo opšte pravilo koje je prvo navedeno.
Slede primeri koji prikazuju kombinovanje rezolucije sa pravilom paramodulacije.

Primer 1

Dokazati da je sledeća formula valjana u jednakosnoj logici:
(Ax)(Ay)(Az)(f(f(x,y),z) = f(x,f(y,z))) /\
(Ax)(n(x) <=> (Ay)(f(x,y) = y /\ f(y,x) = y)) =>
(Ax)(Ay)(n(x) /\ n(y) => x = y)
Rešenje:
NEGACIJA:
(Ax)(Ay)(Az)(f(f(x,y),z) = f(x,f(y,z))) /\
(Ax)(n(x) <=> (Ay)(f(x,y) = y /\ f(y,x) = y)) /\
(Ex)(Ey)(n(x) /\ n(y) /\ x != y)
NNF:
(Ax)(Ay)(Az)(f(f(x,y),z) = f(x,f(y,z))) /\
(Ax)(~n(x) \/ (Ay)(f(x,y) = y /\ f(y,x) = y) /\
n(x) \/ (Ey)(f(x,y) != y \/ f(y,x) != y)) /\
(Ex)(Ey)(n(x) /\ n(y) /\ x != y)
PRENEX:
(Ex)(Ey)(Au)(Ev)(Aw)(Az)
(f(f(u,w),z) = f(u,f(w,z))) /\
(~n(u) \/ (f(u,w) = w /\ f(w,u) = w) /\
n(u) \/ (f(u,v) != v \/ f(v,u) != v)) /\
(n(x) /\ n(y) /\ x != y)
SKOLEMIZACIJA:
(Au)(Aw)(Az)
(f(f(u,w),z) = f(u,f(w,z))) /\

(~n(u) \/ (f(u,w) = w /\ f(w,u) = w) /\
n(u) \/ (f(u,g(u)) != g(u) \/ f(g(u),u) != g(u))) /\
(n(a) /\ n(b) /\ a != b)
KLAUZE:
1) f(f(u,w),z) = f(u,f(w,z))
2) ~n(u) \/ f(u,w) = w
3) ~n(u) \/ f(w,u) = w
4) n(u) \/ f(u,g(u)) != g(u) \/ f(g(u),u) != g(u)
5) n(a)
6) n(b)
7) a != b
REZOLUCIJA SA PARAMODULACIJOM:
8) f(a, w) = w (2, 5) u -> a
9) f(w, a) = w (3, 5) u -> a
10) f(b, w) = w (2, 6) u -> b
11) f(w, b) = w (3, 6) u -> b
12) a = b (paramodulacija, 11, 8):

pošto obe klauze sadrže promenljivu *w* prvo vršimo preimenovanje i dobijamo klauze:

- $f(w_1, b) = w_1$ (koristili smo w -> w_1)
- $f(a, w_2) = w_2$ (koristili smo w -> w_2)

nakon toga u donjoj klauzi zamenjujemo w_1 i primenjujemo najopštiji unifikator za termove $f(w_1, b)$ i $f(a, w_2)$ čime dobijamo:

$(w_1 = w_2)[w_1 \to a, w_2 \to b] \quad \to \quad a = b$

13) [] (7, 12)

Primer 2

Dokazati da je sledeća formula valjana u jednakosnoj logici:
(Ax)(Ay)(Az)(f(f(x,y),z) = f(x,f(y,z))) /\
(Ax)(f(e,x) = x /\ f(x,e) = x) /\
(Ax)(Ay)(i(x, y) <=> f(x,y) = e /\ f(y,x) = e) =>
(Ax)(Ay)(Az)(i(x,y) /\ i(x,z) => y = z)
Rešenje:
NEGACIJA:
(Ax)(Ay)(Az)(f(f(x,y),z) = f(x,f(y,z))) /\
(Ax)(f(e,x) = x /\ f(x,e) = x) /\
(Ax)(Ay)(i(x, y) <=> f(x,y) = e /\ f(y,x) = e) /\
(Ex)(Ey)(Ez)(i(x,y) /\ i(x,z) /\ y != z)
NNF:
(Ax)(Ay)(Az)(f(f(x,y),z) = f(x,f(y,z))) /\
(Ax)(f(e,x) = x /\ f(x,e) = x) /\
(Ax)(Ay)(~i(x, y) \/ (f(x,y) = e /\ f(y,x) = e) /\
i(x, y) \/ f(x,y) != e \/ f(y,x) != e) /\
(Ex)(Ey)(Ez)(i(x,y) /\ i(x,z) /\ y != z)
PRENEX:
(Ex)(Ey)(Ez)(Au)(Av)(Aw)
(f(f(u,v),w) = f(u,f(v,w))) /\

(f(e,u) = u /\ f(u,e) = u) /\
(~i(u, v) \/ (f(u,v) = e /\ f(v,u) = e) /\
i(u, v) \/ f(u,v) != e \/ f(v,u) != e) /\
(i(x,y) /\ i(x,z) /\ y != z)
SKOLEMIZACIJA:
(Au)(Av)(Aw)
(f(f(u,v),w) = f(u,f(v,w))) /\
(f(e,u) = u /\ f(u,e) = u) /\
(~i(u, v) \/ (f(u,v) = e /\ f(v,u) = e) /\
i(u, v) \/ f(u,v) != e \/ f(v,u) != e) /\
(i(a,b) /\ i(a,c) /\ b != c)
KLAUZE:
1) f(f(u,v),w) = f(u,f(v,w))
2) f(e,u) = u
3) f(u,e) = u
4) ~i(u, v) \/ f(u,v) = e
5) ~i(u, v) \/ f(v,u) = e
6) i(u, v) \/ f(u,v) != e \/ f(v,u) != e
7) i(a,b)
8) i(a,c)
9) b != c
REZOLUCIJA SA PARAMODULACIJOM:

10) f(a, b) = e	(4,7)
11) f(b, a) = e	(5,7)
12) f(a, c) = e	(4,8)
13) f(c, a) = e	(5,8)
14) f(e,w) = f(c, f(a, w))	(paramodulacija, 13, 1)
15) f(e,b) = f(c, e)	(paramodulacija, 10, 14)
16) b = f(c,e)	(paramodulacija, 2, 15)
17) b = c	(paramodulacija, 3, 16)
18) []	(9, 17)

Implementacija

U nastavku ove sekcije sledi implementacija algoritma binarne rezolucije sa grupisanjem. Prvo su navedene implementacije gradivnih blokova, odnosno supstitucije i unifikacije pa je na kraju dat i sam algoritam rezolucije.

Supstitucija

```
Term VariableTerm::substitute(const Substitution &s) const
{
   for (const auto & varTermPair : s)
   {
       if (m_var == varTermPair.first)
       {
           return varTermPair.second;
```

```
        }
    }
    return std::const_pointer_cast<BaseTerm>(shared_from_this());
}

Term FunctionTerm::substitute(const Substitution &s) const
{
    std::vector<Term> modifiedTerms;
    modifiedTerms.reserve(m_terms.size());
    for (const Term & t : m_terms)
    {
        modifiedTerms.push_back(t->substitute(s));
    }
    return std::make_shared<FunctionTerm>(m_signature, m_symbol, modifiedTerms);
}

Formula BaseFormula::substitute(const Substitution &s) const
{
    UNUSED_ARG(s);
    return std::const_pointer_cast<BaseFormula>(shared_from_this());
}

Formula Atom::substitute(const Substitution &s) const
{
    std::vector<Term> modifiedTerms;
    modifiedTerms.reserve(m_terms.size());
    for (const Term &t : m_terms)
    {
        modifiedTerms.push_back(t->substitute(s));
    }
    return std::make_shared<Atom>(m_signature, m_symbol, modifiedTerms);
}

Formula Not::substitute(const Substitution &s) const
{
    return std::make_shared<Not>(m_op->substitute(s));
}

Formula And::substitute(const Substitution &s) const
{
    return std::make_shared<And>(m_op1->substitute(s), m_op2->substitute(s));
}

Formula Or::substitute(const Substitution &s) const
{
    return std::make_shared<Or>(m_op1->substitute(s), m_op2->substitute(s));
}

Formula Imp::substitute(const Substitution &s) const
{
    return std::make_shared<Imp>(m_op1->substitute(s), m_op2->substitute(s));
```

```
}

Formula Iff::substitute(const Substitution &s) const
{
    return std::make_shared<Iff>(m_op1->substitute(s), m_op2->substitute(s));
}

template <typename Derived>
Formula Quantifier::substituteImpl(const Substitution &s) const
{
    /*
     * Izdvajamo sve parove koji NE slikaju promenljivu
     * jednaku kvantifikovanoj promenljivoj
     */
    Substitution sCpy;
    std::vector<Term> termsContainingQuantVar;
    for (const auto &varTermPair : s)
    {
        if (varTermPair.first != m_var)
        {
            sCpy[varTermPair.first] = varTermPair.second;

            /* Ako term sadrzi kvantifikovanu promenljivu moramo da je preimenujemo */
            if (varTermPair.second->hasVariable(m_var))
            {
                termsContainingQuantVar.push_back(varTermPair.second);
            }
        }
    }

    /* Ako je skup prezivelih parova prazan vracamo nasu originalnu formulu */
    if (sCpy.empty())
    {
        return std::const_pointer_cast<BaseFormula>(shared_from_this());
    }

    /* Ako neki od termova sadrzi kvantifikovanu promenljivu vrsimo preimenovanje */
    if (!termsContainingQuantVar.empty())
    {
        Variable renamed = getUniqueVarName(m_op, termsContainingQuantVar);
        Formula opWithRenamedVar = m_op->substitute(m_var,

std::make_shared<VariableTerm>(renamed));
        return std::make_shared<Derived>(renamed, opWithRenamedVar->substitute(sCpy));
    }
    else /* Primenimo izmenjenu supstituciju direktno na potformulu */
    {
        return std::make_shared<Derived>(m_var, m_op->substitute(sCpy));
    }
}
```

```
Formula Exists::substitute(const Substitution &s) const
{
   return substituteImpl<Exists>(s);
}

Formula Forall::substitute(const Substitution &s) const
{
   return substituteImpl<Forall>(s);
}
```

Unifikacija

unification.h

```
#ifndef UNIFICATION_H
#define UNIFICATION_H

#include "common.h"
#include "base_term.h"
#include "base_formula.h"

#include <vector>
#include <experimental/optional>
#include <iostream>

/**
* Niz parova termova koje cemo koristiti u supstituciji
*/
using TermPairs = std::vector<std::pair<Term, Term>>;

/**
* Ako je skup termova unifikabilan vracamo supstituciju inace nista,
* zbog toga nam treba opciona substitucija
*/
using OptionalSubstitution = std::experimental::optional<Substitution>;

/**
* @brief unify - unifikuje skup parova termova ako je to moguce
* @param termPairs - parovi termova koje treba unifikovati
* @return najopstiji unifikator
*/
OptionalSubstitution unify(const TermPairs &termPairs);

/**
* @brief unify - unifikuje skup parova termova ako je to moguce
* @param termPairs - parovi termova koje treba unifikovati
* @param s - referenca na supstituciju koju popunjavamo ako su unifikabilni
* @return true ako su parovi unifikabilni, false inace
*/
bool unify(const TermPairs &termPairs, Substitution &s);
```

```
/**
* @brief operator << - ispisuje supstituciju u citljivom formatu
* @param out - stream u koji se vrsi ispis
* @param s - supstitucija koja se ispisuje
* @return referencu na izmenjeni stream
*/
std::ostream& operator<<(std::ostream &out, const Substitution &s);

/**
* @brief operator << - ispisuje parove termova u citljivom formatu
* @param out - stream u koji se vrsi ispis
* @param pairs - parovi koji se ispisuju
* @return referencu na izmenjeni stream
*/
std::ostream& operator<<(std::ostream &out, const TermPairs &pairs);

#endif // UNIFICATION_H
```

unification.cpp

```
#include "unification.h"
#include "variable_term.h"
#include "function_term.h"

#include <algorithm>

static void factoring(TermPairs &termPairs)
{
   /* Za svaki par termova proveravamo da li postoji par koji mu je jednak */
   for (size_t i = 0; i < termPairs.size(); ++i)
   {
       for (size_t j = i+1; j < termPairs.size(); )
       {
           /* Ako su parovi jednaki izbacujemo par indeksiran sa j */
           if (termPairs[i].first->equalTo(termPairs[j].first) &&
                   termPairs[i].second->equalTo(termPairs[j].second))
           {
               /* Zamena sa krajnjim elementom i izbacivanje
        * poslednjeg elementa nakon zamene */
               std::swap(termPairs[j], termPairs.back());
               termPairs.pop_back();
           }
           else /* Ukoliko parovi nisu isti prelazimo na sledeci par */
           {
               ++j;
           }
       }
   }
}

static void tautology(TermPairs &termPairs)
{
```

```
    /* Prolazimo kroz sve parove i eliminisemo one koji sadrze 2 ista terma */
    for (size_t i = 0; i < termPairs.size(); )
    {
        if (termPairs[i].first->equalTo(termPairs[i].second))
        {
            std::swap(termPairs[i], termPairs.back());
            termPairs.pop_back();
        }
        else
        {
            ++i;
        }
    }
}

static bool orientation(TermPairs &termPairs)
{
    /* Za svaki par termova proveravamo da li je drugi promenljiva a prvi ne,
   * ako to jeste slucaj menjamo im mesta
   */
    bool change = false;
    for (auto &termPair : termPairs)
    {
        if (dynamic_cast<VariableTerm*>(termPair.second.get()) &&
                !dynamic_cast<VariableTerm*>(termPair.first.get()))
        {
            std::swap(termPair.first, termPair.second);
            change = true;
        }
    }
    return change;
}

static bool decomposition(TermPairs &termPairs, bool &collision)
{
    /* Za sve parove */
    bool change = false;
    for (size_t i = 0; i < termPairs.size();)
    {
        /* Proveravamo da li su oba clana para funkcijski termovi */
        FunctionTerm *first = dynamic_cast<FunctionTerm*>(termPairs[i].first.get());
        FunctionTerm *second = dynamic_cast<FunctionTerm*>(termPairs[i].second.get());
        if (first && second)
        {
            /* Ako im se simboli razlikuju unifikacija nije uspela */
            if (first->symbol() != second->symbol())
            {
                collision = true;
                return false;
            }
            else
```

```
            {
                /* Simboli su im isti, dodajemo u listu parova parove operanada */
                for (size_t j =0; j < first->operands().size(); ++j)
                {
                    termPairs.emplace_back(first->operands()[j], second->operands()[j]);
                }

                /* Brisemo tekuci par nakon dekompozicije */
                std::swap(termPairs[i], termPairs.back());
                termPairs.pop_back();
                change = true;
            }
        }
        else
        {
            ++i;
        }
    }

    return change;
}

static bool application(TermPairs &termPairs, bool &cycle)
{
    /* Primenjujemo supstituciju v->t za sve parove oblika <v, t>,
     * originalni par se eliminise a supstitucija se primenjuje na
     * sve ostale parove
     */
    bool change = false;
    for (size_t i = 0; i < termPairs.size(); ++i)
    {
        /* Proveravamo da li je prvi clan para promenljiva */
        VariableTerm *first = dynamic_cast<VariableTerm*>(termPairs[i].first.get());
        Term second = termPairs[i].second;
        if (first)
        {
            /* Ako drugi clan para tj. term sadrzi promenljivu koja je prvi clan
             * unifikacija nije uspela
             */
            if (second->hasVariable(first->variable()))
            {
                cycle = true;
                return false;
            }
            else
            {
                /* Za sve parove termova osim tekuceg para primeni supstituciju */
                for (size_t j = 0; j < termPairs.size(); ++j)
                {
                    auto &tpJ = termPairs[j];
                    if (i != j)
```

```
                    {
                        /* Supstituciju primenjujemo na oba clana para termova */
                        if (tpJ.first->hasVariable(first->variable()))
                        {
                            tpJ.first = tpJ.first->substitute(first->variable(), second);
                            change = true;
                        }
                        if (tpJ.second->hasVariable(first->variable()))
                        {
                            tpJ.second = tpJ.second->substitute(first->variable(),
second);
                            change = true;
                        }
                    }
                }
            }
        }
    }

    return change;
}

static bool unify(TermPairs &termPairs)
{
    bool repeat = false;
    bool cycle = false;
    bool collision = false;
    do {

        factoring(termPairs);
        tautology(termPairs);
        repeat = orientation(termPairs) ||
                decomposition(termPairs, collision) ||
                application(termPairs, cycle);

        if (collision || cycle)
        {
            return false;
        }
    } while (repeat);

    return true;
}

OptionalSubstitution unify(const TermPairs &termPairs)
{
    /* Kopiramo skup termova da bismo mogli da ga menjamo i unifikujemo  */
    TermPairs cpyPairs = termPairs;
    if (!unify(cpyPairs))
    {
        return {};
```

```
    }

    /* Iz skupa parova izvlacimo supstituciju */
    Substitution s;
    for (const auto &termsPair : cpyPairs)
    {
        const VariableTerm *vterm = static_cast<const
VariableTerm*>(termsPair.first.get());
        s[vterm->variable()] = termsPair.second;
    }

    return s;
}

std::ostream &operator<<(std::ostream &out, const Substitution &s)
{
    out << "[\t";
    for (const auto &varTermPair : s)
    {
        out << " " << varTermPair.first << "->" << varTermPair.second << "\t";
    }
    return out << "]";
}

std::ostream &operator<<(std::ostream &out, const TermPairs &pairs)
{
    out << "{ ";
    for (const auto &termPair : pairs)
    {
        out << "(" << termPair.first << "," << termPair.second << ") ";
    }
    return out << "}";
}

bool unify(const TermPairs &termPairs, Substitution &s)
{
    TermPairs cpyPairs = termPairs;
    if (!unify(cpyPairs))
    {
        return false;
    }

    /* Iz skupa parova izvlacimo supstituciju */
    s.clear();
    for (const auto &termsPair : cpyPairs)
    {
        const VariableTerm *vterm = static_cast<const
VariableTerm*>(termsPair.first.get());
        s[vterm->variable()] = termsPair.second;
    }
```

```
    return true;
}
```

Rezolucija

resolution.h

```
#ifndef RESOLUTION_H
#define RESOLUTION_H

#include "base_formula.h"

#include <vector>
#include <iostream>

/**
* Definicija tipova za klauze i KNF
*/
using Clause = std::vector<Formula>;
using CNF = std::vector<Clause>;

/**
* @brief resolution - algoritam rezolucije
* @details Algoritam rezolucije je implementiran kao binarna rezolucija sa grupisanjem.
* Iako je ovo potpun sistem, postoje formule za koje se algoritam ne zaustavlja jer
logika
* prvog reda nije odluciva.
* @param cnf - ulazna formula u KNF-u
* @return true ako je formula zadovoljiva, false inace
*/
bool resolution(const CNF &cnf);

/**
* @brief operator << - ispisuje KNF formulu u citljivom formatu
* @param out - stream u koji se ispisuje
* @param cnf - formula koja se ispisuje
* @return referencu na izmenjeni stream
*/
std::ostream& operator<<(std::ostream &out, const CNF &cnf);

#endif // RESOLUTION_H
```

resolution.cpp

```
#include "resolution.h"
#include "first_order_logic.h"
#include "unification.h"

#include <algorithm>
#include <iterator>
```

```
static OptionalSubstitution unify(const Atom *a1, const Atom *a2)
{
   /**
    * Ovo je pomocna funkcija u kojoj vadimo parove termova iz atoma
    * kako bismo probali da unifikujemo 2 atoma.
    */

   if (a1->symbol() != a2->symbol())
   {
       return {};
   }

   TermPairs tpairs;
   const std::vector<Term> &ops1 = a1->operands();
   const std::vector<Term> &ops2 = a2->operands();
   tpairs.reserve(ops1.size());
   for (size_t i = 0; i < ops1.size(); ++i)
   {
       tpairs.emplace_back(ops1[i], ops2[i]);
   }
   return unify(tpairs);
}

static bool clauseExists(const CNF &cnf, const Clause &c)
{
   /**
    * Uslov koji nas zanima je malo labaviji nego da postoji bas takva klauza 'c'
    * u formuli 'cnf'. Posmatrajmo klauze:
    * c1 = (p \/ q \/ r) i c2 = (p \/ q)
    * Ukoliko se c2 nalazi u posmatranoj formuli, nema puno smisla dodavati c1
    * kao novu klauzu. Ako je c2 zadovoljena to znaci da je bar jedan od literala p, q
    * tacan, sto dalje dovodi do toga da je c1 takodje zadovoljena. Klauza c1 nam
    * ne daje nikakve nove restrikcije i nema smisla da je dodajemo (mozemo reci da
    * je c2 sadrzana u c1).
    */

   /* Za sve klauze posmatrane formule proveravamo sadrzanost sa prosledjenom klauzom */
   for (const auto &c1 : cnf)
   {
       /* Ako se svi literali 'c1' nalaze u 'c' smatramo da klauza vec postoji */
       bool subsumed = true;
       for (const auto &l : c1)
       {
           if (std::find(c.cbegin(), c.cend(), l) == c.cend())
           {
               subsumed = false;
               break;
           }
       }

       if (subsumed)
```

```
        {
            return true;
        }
    }
    return false;
}

static bool clauseTautology(const Clause &c)
{
    /* Klauze ja tautologija ako sadrzi suprotne literale */
    for (const auto &l : c)
    {
        /* Ako je u pitanju atom, trazimo njegovu negaciju, inace operand Not-a */
        Formula opositeL;
        const Atom *a = dynamic_cast<const Atom*>(l.get());
        if (a)
        {
            opositeL = std::make_shared<Not>(l);
        }
        else
        {
            opositeL = static_cast<const Not*>(l.get())->operand();
        }

        if (c.cend() != std::find(c.cbegin(), c.cend(), opositeL))
        {
            return true;
        }
    }
    return false;
}

static bool tryGroupLiterals(CNF &cnf, unsigned idx)
{
    /**
     * Trudimo se da unifikujemo sve parove literala klauze, zato
     * definisemo promenljive da je i-ti literal atom 'ai', da je
     * i-ti literal negacija atoma 'ni' i sl.
     */
    bool ret = false;
    Clause &c = cnf[idx];
    Atom *ai = nullptr, *aj = nullptr;
    Not *ni = nullptr, *nj = nullptr;

    /**
     * Za sve parove literala ukoliko su istog tipa trudimo se da ih unifikujemo
     * */
    for (size_t i = 0; i < c.size(); ++i)
    {
        /* Ako dinamicko kastovanje u Atom* nije uspelo, znaci da je literal sigurno Not
*/
```

```
        ai = dynamic_cast<Atom*>(c[i].get());
        ni = !ai ? static_cast<Not*>(c[i].get()) : nullptr;
        for (size_t j = i+1; j < c.size(); ++j)
        {
            /* Ako dinamicko kastovanje u Atom* nije uspelo, znaci da je literal sigurno
Not */
            aj = dynamic_cast<Atom*>(c[j].get());
            nj = !aj ? static_cast<Not*>(c[j].get()) : nullptr;
            OptionalSubstitution s;
            if (ai && aj)
            {
                s = unify(ai, aj);
            }
            else if (ni && nj)
            {
                s = unify(static_cast<Atom*>(ni->operand().get()),
                          static_cast<Atom*>(nj->operand().get()));
            }

            /**
             * Ako su unifikabilni izbacujemo jedan od dva literala, a na sve ostale
             * primenjujemo supstituciju
             */
            if (s)
            {
                Clause cCpy = cnf[idx];
                std::swap(cCpy[j], cCpy.back());
                cCpy.pop_back();
                for (auto &l : cCpy)
                {
                    l = l->substitute(s.value());
                }

                /* Ako klauza vec postoji ili je tautologija nas skup klauza se sustinski
ne menja */
                if (!clauseExists(cnf, cCpy) && !clauseTautology(cCpy))
                {
                    cnf.push_back(cCpy);
                    ret = true;
                }
            }
        }
    }

    return ret;
}

static bool grouping(CNF &cnf, unsigned &idxLastGrpCl)
{
    /**
     * Na sve klauze pocevsi od poslednje za koju smo to vec radili,
```

```
     * trudimo se da primenimo grupisanje
     */
    bool ret = false;
    while (idxLastGrpCl < cnf.size())
    {
        if (tryGroupLiterals(cnf, idxLastGrpCl++))
        {
            ret = true;
        }
    }
    return ret;
}

static void getClauseVars(const Clause &c, VariablesSet &vset)
{
    for (const auto & l: c)
    {
        l->getVars(vset);
    }
}

static bool clauseHasVar(const Clause &c, const Variable &v)
{
    VariablesSet vset;
    getClauseVars(c, vset);
    return vset.find(v) != vset.cend();
}

static Variable getUniqueVar(const Clause &c1, const Clause &c2)
{
    static unsigned s_UniqueCounter = 0U;
    VariablesSet vset;
    getClauseVars(c1, vset);
    getClauseVars(c2, vset);
    Variable unique;
    do {
        unique = "uv" + std::to_string(s_UniqueCounter++);
    } while (vset.find(unique) != vset.cend());
    return unique;
}

static bool tryResolveClauses(CNF &cnf, unsigned i, unsigned j)
{
    /* Obezbedjujemo korektno (pre)imenovanje promenljivih */
    bool ret = false;
    VariablesSet vset;
    getClauseVars(cnf[i], vset);
    for (const auto &v : vset)
    {
        if (clauseHasVar(cnf[j], v))
        {
```

```
            Variable renamed = getUniqueVar(cnf[i], cnf[j]);
            for (auto &l : cnf[j])
            {
                l = l->substitute(v, std::make_shared<VariableTerm>(renamed));
            }
        }
    }

    /* Za sve parove literala klauza probamo da ih unifikujemo ako nisu istog tipa (Atom
i Not) */
    for (size_t k = 0; k < cnf[i].size(); ++k)
    {
        /* Ako dinamicko kastovanje u Atom* nije uspelo, znaci da je literal sigurno Not
*/
        Atom *ai = dynamic_cast<Atom*>(cnf[i][k].get());
        Not *ni = !ai ? static_cast<Not*>(cnf[i][k].get()) : nullptr;
        for (size_t l = 0; l < cnf[j].size(); ++l)
        {
            /* Ako dinamicko kastovanje u Atom* nije uspelo, znaci da je literal sigurno
Not */
            Atom *aj = dynamic_cast<Atom*>(cnf[j][l].get());
            Not *nj = !aj ? static_cast<Not*>(cnf[j][l].get()) : nullptr;
            OptionalSubstitution s;
            if (ai && nj)
            {
                /* Unifikaciju vrsimo nad operandom Not-a */
                s = unify(ai, static_cast<Atom*>(nj->operand().get()));

            }
            else if (ni && aj)
            {
                /* Unifikaciju vrsimo nad operandom Not-a */
                s = unify(static_cast<Atom*>(ni->operand().get()), aj);
            }

            /* Ako je unifikacija uspela */
            if (s)
            {
                /* Izbacujemo suprotne literale iz maticnih klauza */
                Clause cnfCpyI = cnf[i];
                Clause cnfCpyJ = cnf[j];
                std::swap(cnfCpyI[k], cnfCpyI.back());
                cnfCpyI.pop_back();
                std::swap(cnfCpyJ[l], cnfCpyJ.back());
                cnfCpyJ.pop_back();

                /* Kopiramo preostale literale iz obe klauze u rezolventu,
                 * primenjujuci supstituciju usput */
                Clause resolvent;
                resolvent.reserve(cnfCpyI.size() + cnfCpyJ.size());
                std::transform(cnfCpyI.cbegin(),
```

```
                                cnfCpyI.cend(),
                                std::back_inserter(resolvent),
                                [&](const Formula &l) {
                        return l->substitute(s.value());
                    });
                    std::transform(cnfCpyJ.cbegin(),
                                cnfCpyJ.cend(),
                                std::back_inserter(resolvent),
                                [&](const Formula &l) {
                        return l->substitute(s.value());
                    });

                    /* Ako je rezolventa tautologija ili smo vec izveli takvu klauzu
ignorisemo je */
                    if (!clauseTautology(resolvent) && !clauseExists(cnf, resolvent))
                    {
                        cnf.push_back(resolvent);
                        ret = true;
                    }
                }
            }
        }

        return ret;
}

static bool resolventFound(CNF &cnf, unsigned &idxPrev, unsigned &idxCurr)
{
    /* Za sve parove klauza za koje rezolucija nije primenjena do sada, primeni je*/
    bool ret = false;
    while (idxCurr < cnf.size())
    {
        if (tryResolveClauses(cnf, idxPrev, idxCurr))
        {
            ret = true;
        }

        /* Ako je prethodna klauza nije dosla do tekuce, pomeri indeks prethodne ka kraju
*/
        if (idxPrev < idxCurr - 1)
        {
            ++idxPrev;
        }
        else /* Inace, pomeri indeks tekuce klauze ka kraju i pocni ispocetka */
        {
            idxPrev = 0;
            ++idxCurr;
        }
    }

    return ret;
```

```
}

bool resolution(const CNF &cnf)
{
   /**
    * Pravimo kopiju ulazne formule zbog modifikacija koje cemo vrsiti. Uvodimo indekse
    * za poslednju klauzu na koju smo primenili grupisanje 'idxLastGrpCl' kao i
promenljive
    * za pracenje na koje smo klauze primenili pravilo rezolucije 'idxPrev' i 'idxCurr'.
    * Ideja je da rezolviramo klauze sa desna na levo. Na primer klauze 1 i 0, zatim 2 i
0,
    * zatim 2 i 1, zatim 3 i 0, zatim 3 i 1, zatim 3 i 2 itd. Razlog za ovo je sto
stalno
    * dodajemo nove klauze (rezolvente ili rezultate grupisanja) i na ovaj nacin
izbegavamo
    * pozivanje rezolucije vise puta za iste klauze.
    */
   CNF cpyCnf = cnf;
   unsigned idxLastGrpCl = 0;
   unsigned idxPrev = 0;
   unsigned idxCurr = 1;

   /**
    * Dok se skup klauza menja, proveravamo da li smo izveli praznu klauzu
    */
   while (grouping(cpyCnf, idxLastGrpCl) || resolventFound(cpyCnf, idxPrev, idxCurr))
   {
       if (cpyCnf.cend() != std::find_if(cpyCnf.cbegin(),
                                         cpyCnf.cend(),
                                         [=](const Clause &c)
       { return c.empty(); }))
       {
           return false;
       }
   }
   return true;
}

std::ostream &operator<<(std::ostream &out, const CNF &cnf)
{
   out << "[";
   for (const auto &c : cnf)
   {
       out << "[ ";
       std::copy(c.cbegin(), c.cend(), std::ostream_iterator<Formula>(out, " "));
       out << "]";
   }
   return out << "]";
}
```

Vampire - dokazivač za rezonovanje u logici prvog reda

Instalacija

Izvorni kod dokazivača Vampire je slobodno dostupan na adresi https://github.com/vprover/vampire. U trenutku pisanja ovih materijala instalacija putem menadžera paketa (eng. package manager), na primer *apt-get* ili *aptitude* za Ubuntu distribuciju, nije bila moguća.
Sa navedene Veb strane potrebno je skinuti izvorni kod Vampire rešavača. Ukoliko nemate instaliram program *git* imate opciju da skinete .zip datoteku koja sadrži ceo repozitorijum. Otpakujte .zip datoteku na zgodnu lokaciju (u nastavku teksta *vampire_root*). Otvorite terminal i promenite tekući direktorijum na *vampire_root*. Unutar tog direktorijuma biste trebali da imate Makefile. Kod prevodite sa komandom: *make vampire_rel*. Nakon izvršetka ove komande dobićete program *vampire_rel* u okviru direktorijuma gde se nalazi i Makefile. Taj program ćemo pokretati u nastavku kada budemo ispitivali valjanost formula logike prvog reda.

Osnovno o dokazivaču

Dokazivač Vampire očekuje da formule budu zapisane u TPTP formatu. Mi ćemo ovde dati pregled samo najvažnijih sintaksnih pravila, ukoliko čitalac želi da zna više ohrabrujemo ga da poseti Veb stranu http://www.cs.miami.edu/~tptp/.
Formulu navodimo na sledeći način:

fof(ime formule, conjecture, formula).

Na engleskom jeziku *conjecture* označava pretpostavku koju treba dokazati dok je *fof* skraćenica za *first order formula*. Promenljive počinju velikim slovom dok funkcijski i predikatski simboli počinju malim slovom. Vampire podržava rezonovanje u jednakosnoj logici pa samim tim možemo upotrebljavati simbole **=** i **!=** da označimo jednakost, odnosno nejednakost respektivno. Pored ovih simbola koji nisu baš standardni, imamo uobičajene logičke veznike:

- ~ negacija
- & konjunkcija
- | disjunkcija
- => implikacija
- <=> ekvivalencija

Kada želimo da kvantifikujemo promenljive koristimo sledeću sintaksu:

kvantifikator [promenljiva$_1$, …, promenljiva$_n$] : potformula

pri čemu se univerzalni kvantifikator označava sa **!**, a egzistencijalni sa **?**. Dakle, jedan primer kvantifikovane potformule bi bio:

![X] : ![Y] : ?[Z] : (zbir(X, Y) = Z)

gde je interpretacija formule da za svako X i svako Y postoji Z koje je jednako njihovom zbiru. Što se prioriteta operatora tiče, negacija i kvantifikatori imaju veći prioritet nego binarni veznici, dok prioritet između samih binarnih veznika nije definisan već treba koristiti zagrade. Konjunkcija i disjunkcija su levo asocijativni. Na kraju svake formule potrebno je staviti tačku. Jedan primer formule je:

fof(primer, conjecture, ![X, Y] : ? [Z] : (zbir(X, Y) = Z)).

Kada je reč o dokazivanju teorema, odnosno tvrđenja, često se koriste unapred poznate činjenice. Na primer za uobičajeno sabiranje pored ostalih osobina važi da je x+y = y+x. Da bismo zapisali činjenice, odnosno matematičkim jezikom rečeno aksiome, koristimo sledeću sintaksu:

fof(ime formule, axiom, formula).

Dokaz iz aksioma zapisujemo tako što u datoteci prvo navedemo sve aksiome i na dnu datoteke navedemo tvrđenje (conjecture) koje želimo da dokažemo korišćenjem tih aksioma. Vampire rešavač će automatski iskoristiti navedene aksiome gde to bude bilo potrebno.

Kao sam ishod rešavanja Vampire vraća *Refutation found* ukoliko je uspešno izveo praznu klauzu, ili *Satisfiable* ukoliko je prosleđena formula zadovoljiva.

Poslednja stvar na koju treba obratiti pažnju je da Vampire negira ulaznu formulu. Ovo je logično jer pokazujemo da je formula valjana tako što izvedemo praznu klauzu. Imajući to u vidu ako želimo da pokažemo da je neka formula kontradikcija moramo proslediti njenu negaciju jer će Vampire negirati prosleđenu formulu i dve negacije će se onda ispravno poništiti.

U nastavku slede primeri formula logike prvog reda zapisani u TPTP fomatu. Za ove formule ćemo pokretati Vampire rešavač.

Primeri

Primer 1

U gradu postoji jedan berberin. Svi građani se briju ili sami ili ih berberin brije. Da li ovakav berberin postoji?

Rešenje:

Rečenicu možemo zapisati kao:

Ex citizen(x) /\ Ay (citizen(y) => (shaves(x, y) <=> ~shaves(y, y)))

Želimo da proverimo da li je ova formula zadovoljiva. Ako rešavač izvede praznu klauzu to nije slučaj. S obzirom na to da proveravamo baš ovu formulu, a ne njenu negaciju, mi je zbog Vampire rešavača moramo negirati (podsećanje: Vampire uvek negira ulaznu formulu pa je mi zbog toga negiramo da bi se dve negacije poništile). Formula za datoteku *ulaz.tptp* izgleda ovako:

fof(test, conjecture, ~(?[X] : (citizen(X) & ![Y] : (citizen(Y) => (shaves(X, Y) <=> ~shaves(Y, Y))))).

Rešavač vraća *Refutation found*, dakle formula je nezadovoljiva tj. ovakav berberin ne može da postoji.

Mala digresija, nezadovoljivost potiče od toga što mi tvrdimo za svako *y* što uključuje i naše konkretno *x*. Instanciranjem se dobija: *shaves(x, x) <=> ~shaves(x, x)* što je nezadovoljivo.

Primer 2

Neka je dat proizvoljan monoid, neutral za taj monoid je jedinstven.

Pomoć: Monoid je algebarska struktura koja se sastoji od jedne asocijativne operacije, skupa elemenata (ne nužno konačnog) i jednog elementa koji je neutral za tu operaciju. U odnosu na operaciju i neutral skup je zatvoren. Primer monoida je skup prirodnih brojeva sa sabiranjem pri čemu je nula neutral (x+0 = x).

Rešenje:

Trebamo da zapišemo dve stvari. Prva je definicija samog neutrala, tj. šta znači biti neutral. Druga se odnosi na jedinstvenost neutrala. Što se definicije tiče neutral je element takav da kada se operacija primeni na njega i bilo koji drugi element dobije se taj element. U TPTP formatu zapisujemo:

```
fof(monoid_neutral, conjecture,
(![X] : (neutral(X) <=> ![Y] : ((f(X, Y) = Y) & (f(Y, X) = Y)) )) =>
(![X, Y] : ((neutral(X) & neutral(Y)) => X = Y))).
```

Primer 3

Inverz elementa u proizvoljnom monoidu je jedinstven.
Pomoć: Primenom operacije na element i njegov inverz dobija se neutral u odnosu na tu operaciju.
Rešenje:
Potrebno je definisati značenje inverza i onda dodati tvrdnju da je jedinstven. U definiciji inverza učestvuje neutral što znači da moramo imati i koncept neutrala. Nema potrebe da neutral kao neutral definišemo, možemo fiksirati neutral i iskoristiti simbol konstante za njega. U TPTP formatu zapisujemo (napomena je da je *inverz* predikat koji kaže da li je *y* inverz za *x*):

```
fof(monoid_inverz, conjecture,
(![X]:((f(X, e) = X) & (f(e, X) = X)) &
![X, Y]:(inverz(X, Y)<=>((f(X, Y) = e) & (f(Y, X) = e)))) =>
![X, Y1, Y2]:((inverz(X, Y1) & inverz(X, Y2)) => (Y1 = Y2))).
```

Kada pustimo Vampire nad ovom formulom dobićemo iznenađujući rezultat - Satisfiable umesto Refutation found. Iako nam u prethodnom primeru nije trebala, ostala je još jedna činjenica koju nismo iskoristili, a to je da je operacija u monoidu asocijativna. Kada dodamo to u spisak pretpostavki dobijamo fomulu:

```
fof(monoid_inverz, conjecture,
(![X]:((f(X, e) = X) & (f(e, X) = X)) &
![X, Y]:(inverz(X, Y)<=>((f(X, Y) = e) & (f(Y, X) = e))) &
![X, Y, Z]:(f(X, f(Y, Z)) = f(f(X, Y), Z))) =>
![X, Y1, Y2]:((inverz(X, Y1) & inverz(X, Y2)) => (Y1 = Y2))).
```

i očekivani rezultat - Refutation found (polazna formula je valjana jer je izvedena prazna klauza za negaciju polazne formule).

Primer 4

Neka su date sledeće činjenice:

- Svi političari su lukavi
- Samo pokvareni ljudi su političari

Dokazati da ako postoji bar jedan političar, onda postoji pokvaren čovek koji je lukav.
Rešenje:
Prevodimo govorni jezik na jezik logike prvog reda. Uvodimo relacije *političar*, *pokvaren* i *lukav* koje su tačne ako čovek ima te osobine, odnosno netačne u suprotnom. Korišćenjem ovih relacija zapisujemo:

```
fof(politicari, conjecture,
(
   ![X]:(politicar(X) => lukav(X)) &
   ![X]:(~pokvaren(X) => ~(politicar(X)))
) =>
(
   ?[X]:(politicar(X) => ?[Y]:(pokvaren(Y) & lukav(Y)))
)
).
```

Primer 5

Neka su date sledeće činjenice:

- Janko ima psa
- Svaki vlasnik psa voli životinje
- Osoba koja voli životinje ne može da udari životinju
- Janko ili Marko su udarili mačka koji se zove Garfild
- Svaka mačka je životinja

Dokazati da je Marko udario Garfilda.

Rešenje:

Janko i Marko su nam konkretne osobe, dakle konstante. Isto važi i za Garfilda koji je konkretan mačak. Uvodimo predikate *vlasnik_psa*, *voli_zivotinje*, *udario*, *macka i zivotinja* i zapisujemo:

```
fof(garfild, conjecture,
(
   vlasnik_psa(janko) &
   ![X]:(vlasnik_psa(X) => voli_zivotinje(X)) &
   ![X, Y]:((voli_zivotinje(X) & zivotinja(Y)) => ~udario(X, Y)) &
   (udario(janko, garfild) | udario(marko, garfild)) &
   macka(garfild) &
   ![X]:(macka(X) => zivotinja(X))
) =>
(
   udario(marko, garfild)
)
).
```

Čas 11 - rezonovanje u odabranim teorijama (jednakosna teorija i teorija gustih uređenih Abelovih grupa bez krajnjih tačaka)

Jednakosna teorija i normalni modeli

Na prethodnim časovima uglavnom smo tretirali relacijski simbol **=** na isti način kao i sve druge relacijske simbole - značenje je zavisilo isključivo od interpretacije i nije se podrazumevalo. Međutim simbol **=** je zaista specijalan zato što je često korišćen simbol koji skoro uvek predstavlja relaciju jednakosti na domenu na koji se odnosi. U nastavku ćemo sve modele u kojima se simbol **=** koristi za predstavljanje relacije jednakosti nazivati *normalnim modelima*.

Sama relacija jednakosti je određena sledećim aksiomama:

- $\forall x\,(x = x)$ (refleksivnost)
- $\forall x, y\,(x = y <=> y = x)$ (simetričnost)
- $\forall x, y, z\,((x = y \bigwedge y = z) => (x = z))$ (tranzitivnost)

Pored ovih aksioma često se primenjuju i pravila kongruencije:

- $\forall x_1, \ldots, x_n, y_1,\ldots,y_n\ ((x_1=y_1 /\backslash \ldots /\backslash x_n=y_n) => f(x_1,\ldots,x_n) = f(y_1,\ldots,y_n))$

- $\forall x_1, \ldots, x_n, y_1,\ldots,y_n ((x_1=y_1 /\backslash \ldots /\backslash x_n=y_n) => P(x_1,\ldots,x_n) = P(y_1,\ldots,y_n))$

pri čemu su *f* i *P* proizvoljni funkcijski odnosno predikasti simboli.
Ovakva interpretacija simbola jednakosti će nam omogućiti da izgradimo efikasniju proceduru odlučivanja za odgovarajuće teorije. Za početak ćemo se fokusirati na normalne modele, a zatim ćemo pažnju posvetiti teoriji jednakosti sa neinterpretiranim funkcijskim simbolima (eng. *equality with uninterpreted functions*).

Birkhofov sistem

Kada posmatramo normalne modele možemo koristiti Birkhofov deduktivni sistem koji se sastoji od primene sledećih pravila izvođenja (Δ je neki skup jednakosti):

- $s = t \in \Delta \rightarrow \Delta \vdash s = t$ (ax pravilo)
- $\Delta \vdash t = t$ (refl pravilo)
- $\Delta \vdash s = t \rightarrow \Delta \vdash t = s$ (sim pravilo)
- $\Delta \vdash s = t,\ \Delta \vdash t = u \rightarrow \Delta \vdash s = u$ (trans pravilo)
- $\Delta \vdash s_1 = t_1, \ldots, \Delta \vdash s_n = t_n \rightarrow \Delta \vdash f(s_1, \ldots, s_n) = f(t_1, \ldots, t_n)$ (cong pravilo)
- $\Delta \vdash s = t \rightarrow \Delta \vdash (s = t)[x -> a]$ (inst pravilo)

Nazivi pravila odgovaraju skraćenicama za engleske reči, na primer *cong* pravilo odgovara reči *congruence*, *ax* odgovara reči *axiom* itd.
Prilikom dokazivanja imamo polazni skup jednakosti koji odgovara aksiomama i novu jednakost za koju želimo da pokažemo da se primenom navedenih pravila može izvesti iz početnog skupa. Izvođenje je sintaksno prezapisivanje koje započinjemo počevši od jednakosti koju se trudimo da pokažemo. Dakle, možda ne intuitivno na prvi pogled, dokaz se sprovodi odozdo na gore i generiše se drvo dokaza čiji je čvor jednakost koju treba pokazati, a listovi su jednakosti polaznog skupa.
Procedura je odlučiva ukoliko nije potrebno primeniti instanciranje, a inače nije odlučiva jer ne znamo koje instanciranje treba primeniti. Instanciranje nam ne treba ako su sve jednakosti bazne, preciznije ako nemaju slobodne promenljive.

Dokazivanje u Birkhofovom sistemu

Primer 1

Neka je dat sledeći skup jednakosti Δ u kome su *x*, *y*, i *z* podrazumevano univerzalno kvantifikovane promenljive, a *a*, *b*, *c* i *e* su simboli konstanti:

1. f(f(x,y),z) = f(x, f(y,z))
2. f(e, x) = x, f(x, e) = x
3. f(a, b) = e, f(b, a) = e
4. f(a, c) = e, f(c, a) = e

Korišćenjem Birkhofovih pravila dokazati da važi $\Delta \vdash$ b = c.
Rešenje:
Kao što je već rečeno dokazi u ovom sistemu nisu jednostavni za razviti zbog toga što treba odlučiti kad koje pravili primeniti i koje instanciranje izvršiti. Zbog toga pomaže da se prvo napravi *intuitivni dokaz* tvrđenja koji nam pomaže da razvijemo formalni dokaz. S obzirom na to da se ni *b* ni *c* ne nalaze sa desne strane jednakosti moramo da počnemo od 2. time što kažemo da promenljiva *x* dobija baš vrednost konstante *b*:

b = f(e, b)
f(e, b) = f(f(c, a), b) primenom f(c, a) = e
f(f(c, a), b) = f(c, f(a, b)) iz 1. pri x->c, y->a, z->b
f(c, f(a, b)) = f(c, e) primenom f(a, b) = e
f(c, e) = c primenom 2. pri x->c

Strategija u intuitivnom dokazivanju je bila da uvedemo *c* u tekuću jednakost i da iskoristimo 1. za regrupisanje kako bismo od *a* i *b* dobili *e*. Odatle iz 2. lako dobijamo traženo. Intuitivni dokaz će nam pomoći u izboru podciljeva kad budemo gradili dokaz u Birkhofovom sistemu. Sa leve strane "rampe" pisaćemo jednakosti koje su poznate dok će sa desne strane biti ciljevi koje treba da dokažemo. Početno stanje nam je:

f(f(x,y),z) = f(x, f(y,z))
f(e, x) = x, f(x, e) = x
f(a, b) = e, f(b, a) = e
f(a, c) = e, f(c, a) = e |- b = c TRAN

U intuitivnom dokazu *c* se izvodi iz f(c, e) zbog čega primenjujemo **tranzitivnost** čime dobijamo dva cilja(čitalac treba još jednom da se podseti da se dokazivanje izvodi unazad). Ako imamo jednakost u = w mi tranzitivnost primenjujemo tako što uvodimo neko *v* i dobijamo u = v i v = w. Rezon je sledeći, da bih iz početnog skupa jednakosti dokazao u = w, dokazaću da važi u = v i v = w za neko v. Neodlučivost dolazi iz problema izbora koje *v* izabrati? U našem slučaju tranzitivnost ćemo koristiti da instanciramo sve međujednakosti koje se pojavljuju u intuitivnom dokazu. Rezultat prve primene je:

f(f(x,y),z) = f(x, f(y,z))
f(e, x) = x, f(x, e) = x
f(a, b) = e, f(b, a) = e
f(a, c) = e, f(c, a) = e |- b = f(c, e) TRAN
|- f(c, e) = c

Ponovo posmatrajući intuitivni dokaz vidimo da se f(c, e) dobija iz f(c, f(a, b)) pa opet primenjujemo tranzitivnost na prvi podcilj:

f(f(x,y),z) = f(x, f(y,z))
f(e, x) = x, f(x, e) = x
f(a, b) = e, f(b, a) = e
f(a, c) = e, f(c, a) = e |- b = f(c, f(a, b)) TRAN
|- f(c, f(a, b)) = f(c, e)
|- f(c, e) = c

---

f(f(x,y),z) = f(x, f(y,z))
f(e, x) = x, f(x, e) = x
f(a, b) = e, f(b, a) = e
f(a, c) = e, f(c, a) = e |- b = f(f(c, a), b) TRAN
|- f(f(c, a), b) = f(c, f(a, b))
|- f(c, f(a, b)) = f(c, e)
|- f(c, e) = c

---

f(f(x,y),z) = f(x, f(y,z))
f(e, x) = x, f(x, e) = x
f(a, b) = e, f(b, a) = e

```
f(a, c) = e, f(c, a) = e        |- b = f(e, b)              SYM
                                |- f(e, b) = f(f(c, a), b)
                                |- f(f(c, a), b) = f(c, f(a, b))
                                |- f(c, f(a, b)) = f(c, e)
                                |- f(c, e) = c
--------------------------------------------------------------------------------
f(f(x,y),z) = f(x, f(y,z))
f(e, x) = x, f(x, e) = x
f(a, b) = e, f(b, a) = e
f(a, c) = e, f(c, a) = e        |- f(e, b) = b              INS
                                |- f(e, b) = f(f(c, a), b)
                                |- f(f(c, a), b) = f(c, f(a, b))
                                |- f(c, f(a, b)) = f(c, e)
                                |- f(c, e) = c
--------------------------------------------------------------------------------
f(f(x,y),z) = f(x, f(y,z))
f(e, x) = x, f(x, e) = x
f(a, b) = e, f(b, a) = e
f(a, c) = e, f(c, a) = e        |- f(e, x) = x              ASSU
                                |- f(e, b) = f(f(c, a), b)  CONG
                                |- f(f(c, a), b) = f(c, f(a, b))
                                |- f(c, f(a, b)) = f(c, e)
                                |- f(c, e) = c
--------------------------------------------------------------------------------
```

Da bismo malo skratili notaciju primenićemo pravila i na druge ciljeve osim prvog u listi:

```
f(f(x,y),z) = f(x, f(y,z))
f(e, x) = x, f(x, e) = x
f(a, b) = e, f(b, a) = e
f(a, c) = e, f(c, a) = e        |- e = f(c, a)              SYM, ASS
                                |- b = b                    REFL
                                |- f(f(c, a), b) = f(c, f(a, b))   INST, ASSU
                                |- f(c, f(a, b)) = f(c, e)  CONG, jedino nam još ovaj ostaje
                                |- f(c, e) = c              INST, ASSU
--------------------------------------------------------------------------------
f(f(x,y),z) = f(x, f(y,z))
f(e, x) = x, f(x, e) = x
f(a, b) = e, f(b, a) = e
f(a, c) = e, f(c, a) = e        |- c = c          REFL
                                |- f(a, b) = e    ASSU
```

Dokazali smo sve ciljeve počevši od b = c čime je i početno tvrđenje dokazano. Da rezimiramo, jedna strategija za dokaz u Birkhofovom sistemu je da se prvo izvede intuitivni dokaz, a zatim da se sve međujednakosti dobiju primenom tranzitivnosti i na kraju potrebno je razrešiti sve međuciljeve. Ako je intuitivni dokaz dobar, poslednji korak je direktna primena jednog ili dva pravila na svaki podcilj.

Teorija jednakosti sa neinterpretiranim funkcijskim simbolima (eng. equality with uninterpreted functions)

Videli smo da Birkhofova pravila daju negu vrstu procedure odlučivanja pri čemu je instanciranje prilikom primene pravila problem koji nije odlučiv. Ispostavlja se da ipak možemo definisati proceduru odlučivanja za jedan podskup formula. Ako imamo formulu bez slobodnih promenljivih u kojoj je jedini predikatski simbol **=**, pri čemu formula može sadržati razne funkcijske simbole, tada je moguće definisati proceduru odlučivanja zasnovanu na *kongruentnom zatvorenju*.
Formalno, relacija **~** je kongruencija na skupu **D** u odnosu na jezik **L** ako je relacija ekvivalencije na domenu **D** saglasna (kongruentna) sa svim funkcijskim simbolima **f** jezika **L**, odnosno:

$$s_1 \sim t_1, ..., s_n \sim t_n \rightarrow f(s_1, ..., s_n) \sim f(t_1, ..., t_n)$$

Kongruetno zatvorenje relacije je najmanja kongruencija koja sadrži polaznu relaciju.
Intuitivno gledano, recimo da imamo skup termova koji posmatramo i dati skup jednakosti među tim termovima. Smatrajmo da imamo neku proceduru kojom smo sve međusobno jednake termove grupisali u isti skup koji ne sadrži drugi termove osim ovih. Dakle, u jednom ovakvom skupu će se naći i termovi koji su jednaki, a ne pojavljuju se direktno u početnim jednakostima. Na primer za a = b, b = c jasno je da važi i a = c zbog tranzitivnosti pa bi se recimo formirao skup oblika {..., a, b, c, ...}. Kada formiramo ove skupove koji su disjunktni i kojih će potencijalno biti više, skup koji sadrži sve ovakve skupove je kongruentno zatvorenje (skup skupova). Za svaka dva terma koji se nalaze u okviru nekog od elemenata kongruentnog zatvorenja (elementi zatvorenja su skupovi) smatramo da su jednaki.
Postupak odlučivanja u teoriji jednakosti sa neinterpretiranim funkcijskim simbolima (u nastavku EUF) se izvodi po sledećim koracima:

- Ulaz je formula oblika $\forall x_1 ... x_n P(x_1, ..., x_n)$ pri čemu P ne sadrži predikatske simbole osim **=** i ne sadrži kvantifikatore (može sadržati proizvoljne funkcijske simbole)
- Na ulaznu formulu primenjujemo negaciju i skolemizaciju (pokazujemo da je negacija nezadovoljiva iz čega zaključujemo da je polazna formula valjana)
- Prevodimo formulu u DNF - svaka od konjunkcija sadrži samo jednakosti i nejednakosti
- Za svaku konjunkciju :
    - Računamo kongruentno zatvorenje na osnovu jednakosti koje se u njoj javljaju i termova uključujući i njihove podtermove
    - Konjunkcija je zadovoljiva akko za svaki par termova koji formiraju nejednakost važi da se ne nalaze u istoj klasi (u istom skupu koji je element kongruentnog zatvorenja)
- Ako je nijedna konjunkcija nije zadovoljiva početna formula je valjana

Ideja predstavljene procedure je sledeća, ako u okviru bilo koje konjunkcije nema kontradikcije između jednakosti i nejednakosti koje se u njoj javljaju, onda je ta konjunkcija zadovoljiva. Kongruentno zatvorenje računamo jer nam ono daje informaciju o svim mogućim jednakostima između termova, dakle ne samo o direktnim jednakostima koje se javljaju kao deo konjunkcije.
Na kraju, potrebno je definisati algoritam za računanje kongruentnog zatvorenja - jedan od algoritama nosi naziv Nelson-Open algoritam.

Nelson-Open procedura

Podsetimo se, posmatramo konjunkciju oblika:

$$s_1=t_1 \wedge ... \wedge s_n=t_n \wedge s_1' \neq t_1' \wedge ... \wedge s_m' \neq t_m'$$

Formiramo skup E = {s_1=t_1,…, s_n=t_n}. Uvodimo skup T kao skup zatvoren za podtermove koji sadrži bazne termove i možda još neke termove i njihove podtermove. Definišemo skupove i funkcije:

- *use(t)* - skup svih termova čiji je **direktan** podterm term *t*
- *find(t)* - funkcija koja vraća kanonskog predstavnika klase ekvivalencije kojoj pripada term *t*
- *union(s, t)* - funkcija koja spaja klase ekvivalencija za *s* i *t*
- *cong(s, t)* - funkcija koja vraća *true* ako su *s* i *t* kongruentni, inače vraća *false*

Ideja algoritma je sledeća, termove koji učestvuju u baznim jednakostima stavljamo u iste klase ekvivalencije. Zatim, recimo da imamo proizvoljne termove t_1 i t_2 koji su jednaki i dodatno još termove s_1 i s_2 koji sadrže kao direktne podtermove termove t_1 i t_2 respektivno. Za ovakve termove s_1 i s_2 ima smisla da proverimo da li su kongruenti i da ih smestimo u istu klasu ekvivalencije ako jesu (naravno, ako već nisu u istoj klasi). Dakle, procedura funkcioniše tako što pođemo od baznih jednakosti i grupišemo u klase ekvivalencije termove od jednostavnijih ka složenijim koji kao direktne podtermove sadrže ove jednostavnije termove. Treba naglasiti da je procedura rekurzivna, pozivamo spajanje u slučaju da su složeniji termovi kongruentni i da nisu u istoj klasi.

S obzirom na to da implementacija ovog algoritma neće biti prikazana, navodimo pseudokod algoritma (koji zbog toga da bude što realističniji liči na Python kod):

```
def ComputeClosure(E, T):
	# Na početku svaki term je predstavnik svoje klase i jedini term u klasi
	for t in T:
		find(t).insert(t)

	# Spajamo termove koji pripadaju baznim jednakostima u istu klasu
	for e in E:
		Merge(e.term1, e.term2) # term1 = term2

def Merge(s, t):
	# Prazni skupovi na početku
	usingS = {} # U ručnim primerima ćemo ove skupove obeležavati sa T_term
	usingT = {}

	# Formiramo skup svih termova koji kao direktan podterm sadrže
	# term koji je u istoj klasi ekvivalencije sa s
	for term in find(s).all():
		usingS.update(use(term)) # dodajemo sve koji koriste term u skup

	# Formiramo skup svih termova koji kao direktan podterm sadrže
	# term koji je u istoj klasi ekvivalencije sa t
	for term in find(t).all():
		usingT.update(use(term)) # dodajemo sve koji koriste term u skup

	# Sada je bezbedno da izvršimo uniju s i t, find(s) će biti jednako find(t)
	union(s, t)

	# Za sve parove termova spajamo ih rekurzivno ako su kongruenti
	# i nisu u istoj klasi
	for termS in usingS:
		for termT in usingT:
			if find(termT) != find(termS) and congruent(termT, termS):
				Merge(termT, termS)
```

U nastavku slede primeri u kojima su koraci algortima ručno primenjeni radi sticanja intuicije i boljeg razumevanja procedure.

Primer 1

Neka je dat skup jednakosti E = { f(a,b) = a, f(b, a) = b }, pokazati da se jednakost:

f(f(a,b),f(b,a)) = a

može izvesti iz početnog skupa jednakosti.

Rešenje:

Prvo formiramo skup termova T od svih termova prisutnih u jednakostima i on izgleda ovako:

T = {a, b, f(a,b), f(b, a), f(f(a, b), f(b, a))}

Potom formiramo *use(term)* skupove:

use(a) = {f(a, b), f(b, a)}

use(b) = {f(a, b), f(b, a)}

use(f(a, b)) = {f(f(a, b), f(b, a))}

use(f(b, a)) = {f(f(a, b), f(b, a))}

use(f(f(a, b), f(b, a))) = {}

Prvi korak u funkciji *ComputeClosure()* je formiranje *find()* skupova gde je svaki term predstavnik svoje klase:

{{a}, {b}, {f(a, b)}, {f(b, a)}, {f(f(a, b), f(b, a))}}

Sada iteriramo po jednakostima, elementima skupa E, i primenjujemo *Merge()*:

f(a, b) = a

Merge(f(a, b), a)

T_a = use(a) = {f(a, b), f(b,a)} jer se u klasi ekvivalencije sa *a* ne nalazi ni jedan drugi term

$T_{f(a, b)}$ = use(f(a, b)) = {f(f(a, b), f(b, a))}, takođe f(a, b) je sam u svojoj klasi ekvivalencije

union(a, f(a, b)) se izvršava kao sledeći korak i daje izmenjene klase ekvivalencije:

{**{a, f(a, b)}**, {b}, {f(b, a)}, {f(f(a, b), f(b, a))}}

Iteriramo po parovima termova iz T_a i $T_{f(a, b)}$:

f(a, b) ? f(f(a, b), f(b, a)) → termovi nisu u istoj klasi, ali podtermovi b i f(b, a) takođe nisu u istoj klasi pa ne pozivamo Merge()

f(b, a) ? f(f(a, b), f(b, a)) → termovi nisu u istoj klasi, ali podtermovi b i f(a, b) takođe nisu u istoj klasi pa ne pozivamo Merge()

f(b, a) = b

Merge(f(b, a), b)

T_b = {f(a, b), f(b, a)}

$T_{f(b, a)}$ = {f(f(a, b), f(b, a))}

union(f(b, a), b) daje nove klase ekvivalencije:

{{a, f(a, b)}, **{b, f(b, a)}**, {f(f(a, b), f(b, a))}}

Za sve parove iz iz T_b i $T_{f(b, a)}$:

f(a, b) ? f(f(a, b), f(b, a)) → funkcijski simboli isti, a i f(a, b) su u istoj klasi, b i f(b, a) su u istoj klasi što znači da su termovi kongruentni, a pošto nisu u istoj klasi zovemo Merge()

Merge(f(a, b), f(f(a, b), f(b, a)))

$T_{f(f(a, b), f(b, a))}$ = {} zbog čega ne moramo da razmatramo uopšte skup $T_{f(a, b)}$ već samo primenjujemo uniju:

union(f(a, b), f(f(a, b), f(b, a))) koja daje:

{**{a, f(a, b), f(f(a, b), f(b, a))}**, {b, f(b, a)}}

f(b, a) ? f(f(a, b), f(b, a)) → podtermovi b i f(a, b) nisu u istoj klasi, dakle termovi nisu Kongruentni pa se ne poziva Merge()

Nema više jednakosti i petlja po elementima skupa E se završava.

Klase ekvivalencije koje smo dobili su:

{{a, f(a, b), f(f(a, b), f(b, a))}, {b, f(b, a)}}

pri čemu odgovaramo na pitanje da li važi ciljna jednakost f(f(a,b),f(b,a)) = a. S obzirom na to da ovi termovi pripadaju istoj klasi ekvivalencije u kongruentom zatvorenju jednakost je dokazana.

Primer 2

Pokazati da je formula logike prvog reda:

$\forall x, y\ ((y = f(x) \bigwedge x = g(y)) => (x = g(f(x))))$

valjana u EUF teoriji.

Rešenje:

Da bismo pokazali valjanost, pokazujemo nezadovoljivost negacije ulazne formule. Prateći korake za rezonovanje u EUF teoriji primenjujemo (koraci su dati u sekciji):

NEGACIJA:

$\neg\forall x, y\ ((y = f(x) \bigwedge x = g(y)) => (x = g(f(x))))$ implikaciju rastavljamo na ~a \/ b, negacijom dobijamo a /\ ~b

$\exists x, y\ ((y = f(x) \bigwedge x = g(y)) \bigwedge x != g(f(x)))$

SKOLEMIZACIJA (formula je već u prenex normalnoj formi):

b = f(a) /\ a = g(b) /\ a != g(f(a))

Nakon skolemizacije formula je automatski u DNF-u, da nije tako morali bismo da prevedemo formulu u DNF. Imamo samo jednu konjunkciju tako da ispitujemo nezadovoljivost samo za nju. Da smo nekim slučajem dobili više konjunkcija morali bismo za svaku konjunkciju da posebno sprovodimo Nelson-Openov algoritam. Za trenutnu formulu formiramo skupove jednakosti i termova:

E = {b = f(a), a = g(b)}

T = {a, b, f(a), g(b), **g(f(a))**} → obavezno dodajemo i termove koji učestvuju u nejednakostima

Formiramo *use* skupove:

use(a) = {f(a)}

use(b) = {g(b)}

use(f(a)) = {g(f(a))}

use(g(b)) = {}

use(g(f(a))) = {}

Klase ekvivalencije:

{{a}, {b}, {f(a)}, {g(b)}, {g(f(a))}}

Iteriramo po jednakostima:

b = f(a)

Merge(b, f(a))

T_b = {g(b)}, $T_{f(a)}$ = {g(f(a))}

union(b, f(a)) → {{a}, **{b, f(a)}**, {g(b)}, {g(f(a))}}

Iteriramo po parovima iz T_b i $T_{f(a)}$:

g(b) ? g(f(a)) → termovi su kongruenti zovemo Merge()

Merge(g(b), g(f(a)))

$T_{g(b)}$ = {} → samo zovemo uniju jer je bar jedan skup prazan

union(g(b), g(f(a))) → {{a}, {b, f(a)}, **{g(b), g(f(a))}**}

a = g(b)

Merge(a, g(b))

$T_{g(b)}$ = use(g(b)) U use(g(f(s))) = {} → samo zovemo uniju jer je bar jedan skup prazan, primetiti da ima više od jednog terma u klasi ekvivalencije, dakle imamo uniju više *use* skupova

union(a, g(b)) → {{b, f(a)}, **{a, g(b), g(f(a))}**}

Nema više parova jednakosti.

S obzirom na to da imamo nejednakost a != g(f(a)), pri čemu se termovi a i g(f(a)) nalaze u istoj klasi ekvivalencije konjunkcija je nezadovoljiva. Pošto su sve konjunkcije nezadovoljive, treba imati u vidu da imamo samo jednu konjunkciju i da je ona nezadovoljiva, onda i i formula nezadovoljiva iz čega zaključujemo da je ulazna formula valjana u EUF teoriji.

Teorija gustih Abelovih grupa bez krajnjih tačaka

Komplikovan naziv ove teorije služi samo da na koncizan način sažme sve osobine ove teorije tako da se dobije relativno kratak naziv. Na primer, model jedne ovakve teorije je dat sa (**R**, +, -, <, =) gde je **R** skup realnih brojeva, + operacija (funkcija) sabiranja, - operacija dobijanja inverza, < relacija “manje od” i = relacija jednakosti. Detaljnije, ako formiramo sledeću konjunkciju pri uobičajenoj interpretaciji sintaksnih simbola kojom je zapisujemo:

x + y < 5 /\

y + z < x + (-15) /\

z + x < -7

pri čemu je verovatno prirodno ovaj problem posmatrati kao problem rešavanja sistema nejednačina.

Da se podsetimo, grupa je Abelova ako je komutativna. Grupa je gusta ako za svaki par vrednosti *x*, *y* postoji *z* čija je vrednost između *x* i *y* u smislu poretka. Krajnje tačke se odnose na to da ne postoji pravi maksimum odnosno minimum. To znači da za svako *x* postoji *y* koje je od njega veće. Takođe za svako *x* postoji *y* koje je od njega manje. Sada bi trebalo da je naziv teorije jasniji, tako da ćemo se posvetiti njenim aksiomama, pri čemu se **=** interpretira kao relacija jednakosti, to jest ograničavamo se samo na normalne modele. Dodatno, signatura sadrži +, -, =, <, 0 gde je 0 simbol za neutral:

1. $\forall x,\ y\ (x + y = y + x)$ [komutativnost]
2. $\forall x, y, z\ ((x + y) + z = x + (y + z)\ )$ [asocijativnost]
3. $\forall x,\ y,\ z\ (x < y \bigwedge y < z => x < z)$ [tranzitivnost]
4. $\forall x\ (x + 0 = x)$ [neutral]
5. $\forall x\ (x + (-x)) = 0$ [inverzni element]
6. $\forall x\ \neg(x < x)$ [antirefleksivnost]
7. $\forall x, y\ (x < y \bigvee y < x \bigvee x = y)$ [totalni poredak]
8. $\forall x, y, x\ (x < y => x + z < y + z)$ [saglasnost sa sabiranjem]
9. $\forall x, y\ (x < y => \exists z\ (x < z \bigwedge z < y))$ [gusta grupa]
10. $\forall x \exists y\ (y < x)$ [ne postoji najmanji element]
11. $\forall x \exists y\ (x < y)$ [ne postoji najveći element]

Dakle, sve strukture koje zadovoljavaju ove aksime predstavljaju modele ove teorije.

Što se tiče problema ispitivanja valjanosti ispostavlja se da postoje određene olakšice vezane za ovu teoriju. Važe sledeće osobine teorije:

- Za svaku formulu ove teorije bez slobodnih promenljivih (*zatvorena formula*) važi da je ili ona valjana u ovoj teoriji ili je njena negacija valjana.
- Postoji procedura odlučivanja koja za svaku zatvorenu formulu ove teorije odlučuje da li je ona valjana - teorija je odlučiva.

Iz prve osobine takođe sledi da je formula valjana u teoriji akko je zadovoljiva u njoj. U nastavku će biti prikazana jedna procedura odlučivanja za ovu teoriju - Furije-Mockinova eliminacija. Za formalan tretman ove teorije konsultovati poglavlje 3.4.3 knjige "Matematička logika u računarstvu" profesora Predraga Janičića.

Furije-Mockinova procedura

Ovo odlučivanje se sprovodi za zatvorene formule. Osnovna ideja procedure je da koristimo nejednakosti da eliminišemo promenljivu po promenljivu. Na primer recimo da imamo nejednakosti oblika:

$$x < y + 5 \text{ i } z - 3 < x$$

eliminaciju promenljive *x* sprovodimo tako što objedinimo nejednakosti i dobijamo:

$$z - 3 < y + 5$$

Treba primetiti da sličnu eliminaciju možemo izvršiti i ako imamo jednu jednakost i jednu nejednakost, malo modifikovan prethodni primer daje:

$$x < y + 5 \text{ i } z - 3 = x \rightarrow z - 3 < y + 5$$

S obzirom na to da nemamo direktno simbol <= u značenju "manje ili jednako" ovu relaciju modelujemo na sledeći način:

$$x <= y \rightarrow x < y \setminus/ x = y$$

dakle koristimo disjunkciju jednakosti i manje od. Jedna od situacija kada nam treba da modelujemo <= je prilikom negacije izraza oblika x < y. Semantički gledano, rezultat ove negacije je da je *y* manje do jednako *x* pri čemu mi koristimo razlaganje navedeno iznad čime se dobija:

$$\sim(x < y) <=> y < x \setminus/ x = y$$

Slično, negacije jednakosti možemo rastaviti na dve nejednakosti:

$$\sim(x = y) \rightarrow x < y \setminus/ y < x$$

Dakle ovim pravilima je data mehanizacija za eliminaciju jedne promenljive. U opštem slučaju mi ćemo imati *n* nejednakosti oblika x < $izraz_i$ i *m* nejednakosti oblika $izraz_j$ < x. Svaka nejednakost gde se *x* nalazi sa leve strane nejednakosti se kombinuje sa nejednakostima gde se *x* nalazi sa desne strane nejednakosti. Ovaj postupa eliminiše promenljivu *x* i uvodi *n*m* novih nejednakosti u opštem slučaju. S obzirom na to da eliminacija jedne promenljive u najgorem slučaju dovodi do kvadratno mnogo novih nejednakosti složenost samog algoritma je eksponencijalna (postoje efikasniji algoritmi).

Prethodna eliminacija je data prilično neformalno radi lakšeg razumevanja i što jednostavnije notacije. Podsetimo se da mi zapravo rezonujemo o zatvorenim formulama, a u proceduri eliminacije se nismo dotakli teme šta se radi sa kvantifikatorima i sa formulom uopšte. Formalno, kao ulaz posmatramo pojednostavljenu formulu u prenex normalnoj formi koja je oblika:

$$Q_1x_1...Q_nx_nQ_{n+1}y(\text{Formula}(x_1,..., x_n, y))$$

Prvi korak je da se osiguramo da je kvantifikator Q_{n+1} koji stoji uz *y* egzistencijalni. Ako je već egzistencijalni ne radimo ništa, dok ako je univerzalni primenjujemo transformaciju:

$$Q_1x_1...Q_nx_n \forall y(\text{Formula}(x_1,..., x_n, y)) \rightarrow Q_1x_1...Q_nx_n \sim \exists y(\sim\text{Formula}(x_1,..., x_n, y))$$

Sledeći cilj je da potformulu ~Formula(x_1,..., x_n, y) prebacimo u DNF. To radimo eliminacijom ekvivalencija i implikacija, spuštanjem negacije na nivo atoma i na kraju primenom De Morganovih

zakona da bi se disjunkcije i konjunkcije ispravno pozicionirale. Recimo da je rezultujuća formula oblika:

$$Q_1x_1...Q_nx_n \sim \exists y(A_1 \vee A_2 \vee \dots \vee A_m)$$

pri čemu su A_i konjunkcije. Egzistencijalni kvantifikator može da "prolazi" kroz disjunkciju i naredni korak je da spustimo ovaj kvantifikator na nivo konjunkcija čime se dobija:

$$Q_1x_1...Q_nx_n \sim (\exists y(A_1) \vee \exists y(A_2) \vee \dots \vee \exists y(A_m))$$

Sada je potrebno iz svake konjunkcije A_i eliminisati promenljivu *y*. Ako konjunkcija A_i ne sadrži *y* tada samo obrišemo kvantifikator. U suprotnom, ako A_i sadrži *y* kombinujemo nejednakosti i jednakosti na već opisan način tako da se ova promenljiva eliminiše. Nakon eliminacije možemo bezbedno obrisati kvantifikator i kao rezultat se dobija:

$$Q_1x_1...Q_nx_n \sim (B_1 \vee B_2 \vee \dots \vee B_m)$$

pri čemu su B_i konjunkcije dobijene od A_i eliminacijom *y* tako da ne sadrže ovu promenljivu. Postupak primenjujemo dok ne eliminišemo sve promenljive x_i. Kada eliminišemo sve promenljive na kraju će nam ostati izrazi iz kojih direktno izvodimo da li je formula valjana ili ne.

Poslednja stvar na koju treba obratiti pažnju je šta radimo kada imamo nejednakosti oblika:

$$4x < 18y \text{ i } 12z < 2x$$

Treba razumeti da je 4x samo skraćeni zapis za x+x+x+x, a slično važi i za 17y, odnosno 12z. Imajući ovo u vidu rezultujuća nejednakost koja se dobija eliminacijom promenljive *x* bi bila:

$$4z < 3y$$

U nastavku sledi par primera u kojima će biti prikazana primena ove procedure.

Primer 1

Dokazati da je data formula teorema teorije gustih Abelovih grupa bez krajnjih tačaka:

$$\forall x, y, z\ (2x < 3y \wedge 3x < 2y \wedge 7y < 5z \Rightarrow 14x < 10z)$$

Rešenje:

Primenjujemo prvi korak postupka i konvertujemo najunutrašnjiji kvantifikator da bude egzistencijalni umesto univerzalni čime dobijamo:

$$\forall x, y \sim \exists z \sim (2x < 3y \wedge 3x < 2y \wedge 7y < 5z \Rightarrow 14x < 10z)$$

$$\forall x, y \sim \exists z\ (2x < 3y \wedge 3x < 2y \wedge 7y < 5z \wedge \sim(14x < 10z))$$

$$\forall x, y \sim \exists z\ (2x < 3y \wedge 3x < 2y \wedge 7y < 5z \wedge (10z < 14x \vee 14x = 10z))$$

$$\forall x, y \sim \exists z\ ((2x < 3y \wedge 3x < 2y \wedge 7y < 5z \wedge 10z < 14x) \vee (2x < 3y \wedge 3x < 2y \wedge 7y < 5z \wedge 14x = 10z))$$

Sada egzistencijalni kvantifikator prolazi kroz disjunkcije i dobijamo dve konjunkcije iz kojih eliminišemo promenljivu *z*, transformišemo prvu:

$$\exists z\ (2x < 3y \wedge 3x < 2y \wedge \underline{7y < 5z \wedge 10z < 14x})$$

$$\exists z\ (2x < 3y \wedge 3x < 2y \wedge y < x) \rightarrow \text{nema } z \text{ brišemo kvantifikator}$$

$$2x < 3y \wedge 3x < 2y \wedge y < x$$

Iz druge se dobija:

$$\exists z\ (2x < 3y \wedge 3x < 2y \wedge \underline{7y < 5z \wedge 14x = 10z})$$

$$\exists z\ (2x < 3y \wedge 3x < 2y \wedge y < x)$$

$$2x < 3y \wedge 3x < 2y \wedge y < x$$

Pošto smo dobili dve iste konjunkcije nema potrebe da ih koristimo obe, pišemo samo jednu i sada formula izgleda ovako:

$$\forall x, y \sim (2x < 3y \wedge 3x < 2y \wedge y < x)$$

Kao što smo eliminisali *z* isto tako eliminišemo i *y*:

$$\forall x, y \sim (2x < 3y \wedge 3x < 2y \wedge y < x)$$

$\forall x \sim \exists y \sim\sim(2x < 3y \wedge 3x < 2y \wedge y < x)$

$\forall x \sim \exists y\ (2x < 3y \wedge 3x < 2y \wedge y < x)$ → ista boja, x je sa iste strane

$\forall x \sim \exists y\ (2x < 3x \wedge 3x < 2x)$

$\forall x \sim (2x < 3x \wedge 3x < 2x)$

Na kraju eliminišemo x:

$\sim \exists x \sim\sim(2x < 3x \wedge 3x < 2x)$

$\sim \exists x\ (2x < 3x \wedge 3x < 2x)$ → skinemo po 2x sa svake strane

$\sim \exists x\ (0 < x \wedge x < 0)$

$\sim \exists x\ (0 < 0)$

$\sim(0 < 0)$

$\sim F$

T

Eliminacijom promenljivih smo došli do toga da je polazna formula zadovoljiva. Zbog osobina teorija, formula koja je zadovoljiva je ujedno i valjana, to jest teorema, čime je tvrđenje pokazano.

Primer 2

Dokazati da je data formula teorema teorije gustih Abelovih grupa bez krajnjih tačaka:

$\forall x, y, z, u\ (x < y \wedge x + y = 2z \wedge y - x = u) \Rightarrow\ z + u > y)$

Rešenje:

Kao i prvom primeru krećemo sa eliminacijom najunutrašnjije kvantifikovane promenljive, u našem slučaju to je u:

$\forall x, y, z \sim \exists u \sim((x < y \wedge x + y = 2z \wedge y - x = u) \Rightarrow\ z + u > y)$

$\forall x, y, z \sim \exists u\ ((x < y \wedge x + y = 2z \wedge y - x = u) \wedge (z + u < y \vee z + u = y))$

$\forall x, y, z \sim \exists u\ ((x < y \wedge x + y = 2z \wedge y - x = u \wedge z + u < y) \vee (x < y \wedge x + y = 2z \wedge y - x = u \wedge z + u = y))$

$\forall x, y, z \sim (\exists u\ (x < y \wedge x + y = 2z \wedge \underline{y - x = u \wedge z + u < y}) \vee \exists u\ (x < y \wedge x + y = 2z \wedge \underline{y - x = u \wedge z + u = y}))$

$\forall x, y, z \sim (\exists u\ (x < y \wedge x + y = 2z \wedge z - x < 0) \vee \exists u\ (x < y \wedge x + y = 2z \wedge z - x < 0))$

$\forall x, y, z \sim((x < y \wedge x + y = 2z \wedge z - x < 0) \vee (x < y \wedge x + y = 2z \wedge z - x = 0))$

$\forall x, y, z \sim((x < y \wedge x + y = 2z \wedge z - x < 0) \vee (x < y \wedge x + y = 2z \wedge z - x = 0))$ → eliminišemo z sledeće

$\forall x, y \sim \exists z \sim\sim((x < y \wedge x + y = 2z \wedge z - x < 0))$

$\forall x, y \sim \exists z\ ((x < y \wedge x + y = 2z \wedge z - x < 0) \vee (x < y \wedge x + y = 2z \wedge z - x = 0))$

$\forall x, y \sim (\exists z\ (x < y \wedge x + y = 2z \wedge z - x < 0) \vee \exists z\ (x < y \wedge x + y = 2z \wedge z - x = 0))$

$\forall x, y \sim((x < y \wedge y < x) \vee (x < y \wedge y = x))$ → eliminišemo y sledeće

$\forall x \sim \exists y \sim\sim((x < y \wedge y < x) \vee (x < y \wedge y = x))$

$\forall x \sim \exists y\ ((x < y \wedge y < x) \vee (x < y \wedge y = x))$

$\forall x \sim (\exists y\ (x < y \wedge y < x) \vee \exists y\ (x < y \wedge y = x))$

$\forall x \sim (\exists y\ (x < x) \vee \exists y\ (x < x))$

$\forall x \sim((x < x) \vee (x < x))$

$\forall x \sim((x < x))$

Nakon ovoga možemo se čak pozvati i na aksiomu teorije antirefleksivnost iz koje vidimo da je formula zadovoljiva odakle je i valjana. Precizan postupak bi eliminisao i promenljivu x i sveo početnu formulu na ~(0 < 0) što je zapravo T.

Čas 12 i 13 - *Satisfiability Modulo Theories* rešavači (SMT rešavači)

Uvod

Kao što smo već videli, za veliki deo potencijalnih primena kodiranje problema u iskaznoj logici nije pravo rešenje i potrebna nam je logika prvog reda. S druge strane logika prvog reda nije odlučiva u opštem slučaju što nam na prvi pogled stvara veliki problem. Ovde treba imati na umu da smo počeli od primena i da imamo olakšanje u vidu toga što nas ne zanima opšti slučaj - zanima nas da rešavamo konkretne probleme.

Uzmimo na primer moderne procesore koji imaju mogućnost preklapanja instrukcija (u istom ciklusu svaki deo procesora izvršava odgovarajući deo neke instrukcije paralelno; instrukcije traju više ciklusa tako da se izvršavanje preklapa). Verifikaciju ispravnog rada ovakvih procesora je moguće modelovati kao EUF formule. EUF predstavlja jednu teoriju koja je zapravo odlučiva za formule bez slobodnih promenljivih (rečenice) i to u polinomijalnom vremenu (Nelson-Open algoritam).

Za slučaj da prethodni primer nije toliko intuitivan posmatrajmo *sisteme u realnom vremenu* (eng. real time systems). U današnje vreme sve popularnija tema je autonomna vožnja koja se bazira na integraciji velikog broja složenih algoritama u odluku šta će automobil sledeće uraditi. Odluka se mora doneti u realnom vremenu, to jest dovoljno brzo da recimo automobil ne udari u pešaka. Sistemi koji treba da rade u realnom vremenu se mogu zapisati *vremenskim automatima* (eng. timed automata), dok se rezonovanje o radu ovih automata može izvesti u skladu sa teorjim *logike razlike* (eng. difference logic) koja sadrži formule oblika:

$S_a + t <= S_b$ → korak *b* može krenuti tek *t* sekundi nakon koraka *a*

$S_a <= S_b + t$ → korak *a* mora da se započne u najviše *t* sekundi nakon *b*

Kako je programiranje još bliža tema čitaocu, posmatrajmo problem modelovanja računarske aritmetike za cele brojeve. Šta se dešava za sledeći fragment koda:

```
int32_t x = 5;
if (x > x + 2147483647) { … }
```

Na prvi pogled izgleda da uslov grananja nikad nije ispunjen jer poredimo 5 > 2147483652 što je zaista netačno. Međutim model celih brojeva nam je pogrešan! Ne možemo da koristimo uobičajenu aritmetiku jer je opseg brojeva u računaru ograničen. Predstavljeni uslov je zapravo uvek ispunjen jer dolazi do prekoračenja sa desne strane nejednakosti i taj zbir zapravo postaje negativan broj, dok je broj 5 očigledno pozitivan. Umesto standardne aritmetike koristi se teorija bitvektora. Ovaj naziv je prirodan jer se brojevi u račinaru zapisuju kao nizovi bitova.

Na samom kraju treba naglasiti da se u praksi često kombinuje više teorija. Nadovezujući se na prethodni primer pogledajmo malo izmenjen kod:

```
int32_t x[2] = {5, 7};
if (x[0] > x[1] + 2147483647) { … }
```

Dakle, ovde imamo sabiranje bitvektora, nejednakost, imamo niz i pristup elementima niza. U realnim scenarijima moguće je da imamo i konjunkciju više uslova koji uključuju i realne brojeve i slično. Do kraja poglavlja koje se odnosi na ovu nastavnu jedinicu upoznaćemo se sa rešavanjem problema koji kombinuju više teorija.

U ovoj sekciji je navedeno par praktičnih primena naizgled čisto teorijskih koncepata za rešavanje konkretnih problema iz industrije. Naravno, ovo nije ni blizu potpun spisak primena.

DPLL(T) algoritam

Postavlja se pitanje kako pristupiti problemu ispitivanja zadovoljivosti složenih formula koje sadrže najrazličitije atome. Jedan primer klauze koja bi mogla da se javi u ovakvoj formuli je:

$$p \vee \neg q \vee a=f(b-c) \vee read(s, f(b-c)) \vee a-g(c) < 7$$

Problem koji razmatramo je problem zadovoljivosti u prisustvu teorija u pozadini (eng. background theories). U nastavku ćemo ovaj problem referisati kao SMT (eng. **S**atisfiability **M**odulo **T**heories), dok ćemo rešavače za ovaj problem zvati SMT rešavači.

Ispostavlja se da je jedan validan pristup da sve atome posmatramo kao da su u pitanju atomi iskazne logike. Dakle svaki atom logike prvog reda zamenimo iskaznim slovom i rešavamo problem iskazne zadovoljivosti za koji imamo efikasne SAT rešavače. Ako je iskazna formula nezadovoljiva, očigledno je i formula logike prvog reda nezadovoljiva. S druge strane ako je iskazna formulacija zadovoljiva, ne znači da je zadovoljiva i početna formula. Valuacija koja zadovoljava iskaznu varijantu postavlja neke atome na tačno, a neke na netačno. Potrebno je proveriti da li je takva dodela u skladu sa teorijom ili kombinacijom teorija kojoj atomi pripadaju. Ako nije, moramo tražiti neku drugu valuaciju koja zadovoljava iskaznu formulu.

U pasusu iznad je data intuicija kako bismo mogli da formiramo proceduru odlučivanja za SMT problem. Trenutne tehnike koje se koriste za rešavanje SMT problema se dele na *gramzive* (eng. eager) i *lenje* (eng. lazy). Predstavićemo ukratko osobine i jednog i drugog pristupa kao i probleme, a onda ćemo se fokusirati na DPLL(T) arhitekturu koja predstavlja osnovu za najefikasnije SMT rešavače današnjice.

Osnovna ideja gramzivog pristupa je da se celokupna formula logike prvog reda prevede u formulu iskazne logike. To znači da se značenja atoma konkretne teorije moraju kodirati iskaznim promenljivama. Podsetimo se na trenutak da se najrazličitije veze mogu kodirati kao SAT problem, jedan primer je predstavljen kroz sudoku igricu u jednom od prethodnih poglavlja. Dakle, u ulaznoj formuli se svi atomi konkretnih teorija kodiraju kombinacijom novih iskaznih promenljivih, zatim se formula prevede u KNF i kao takva se da SAT rešavaču. Prednost ovog pristupa je što se uvek može koristiti najbolji dostupan SAT rešavač nezavisno od samih teorija. Međutim ovaj pristup ima više problema nego prednosti i danas se koristi za usko specijalizovane primene kada formule ne zahtevaju puno različitih teorija u pozadini. Glavni problem je što se za efikasno kodiranje svake od teorija mora razviti specijalizovani efikasan postupak koji je najčešće primenljiv samo na tu teoriju. Efikasna kodiranja postoje za EUF, teoriju celobrojne aritmetike i teoriju nizova.

Komplementaran pristup u odnosu na gramzivu strategiju je da uzmemo početnu formulu, konvertujemo je u DNF i proverimo da li je bar neka od DNF konjunkcija tačna uz pomoć teorijskih rešavača (na primer Nelson-Open za EUF, Furije-Mockin ili simpleks algoritam za realnu aritmetiku itd.). Kao što smo već ranije utvrdili konverzija u DNF je problematična i dolazi do kombinatorne eksplozije zbog čega ovaj pristup nije praktično upotrebljiv.

Lenja strategija se nalazi negde između navedenih pristupa. Od ulazne formule logike prvog reda gradimo formulu iskazne logike takvu da svakom atomu originalne formule odgovara jedno iskazno slovo. Ovako formiranu formulu prosleđujemo SAT rešavaču. Kao što je već navedeno u nekom od gornjih pasusa, ako je formula zadovoljiva SAT rešavač nam vraća valuaciju. Na osnovu valuacije formiramo konjunkciju atoma logike prvog reda to jest originalne formule i prosleđujemo je teorijskom rešavaču koji proverava da li je data konjunkcija u skladu sa teorijom. Ako jeste

originalna formula je zadovoljiva. Ako nije nastavljamo pretragu. Pre nastavka pretrage teorijski rešavač gradi novu klauzu koja je logička posledica teorije i koja sprečava konfliktnu dodelu vrednosti u valuaciji. Dodavanjem ove nove klauze koja je posledica teorije sprečavamo SAT rešavač da ponovo generiše valuaciju koja ima isti konflikt u samoj teoriji. Pažljiv čitalac bi mogao da primeti da u opštem slučaju nakon dodavanja svake nove klauze moramo da pokrećemo SAT rešavač iznova što je neefiksano. Ispostavlja se da možemo da izbegnemo pokretanje iznova ukoliko je algoritam za SAT zasnovan na DPLL proceduri i ako teorijski rešavači imaju određene osobine koje ćemo navesti. Ovakva kombinacija nam daje izuzetno efikasan algoritam odlučivanja. Poboljšanja primenjiva pri lenjoj strategiji su:

- Inkrementalni teorijski rešavač (u nastavku samo T-rešavač) - u DPLL proceduri valuacija koja zadovoljava formulu se gradi korišćenjem parcijalne valuacije. Ideja inkrementalnog T-rešavača je da proverava konzistentnost trenutne parcijalne valuacije sa teorijom prilikom svakog novog postavljanja literala na neku vrednost. S obzirom na to da proveravanje T-konzistentnosti može da bude vremenski zahtevna operacija, često se provera vrši na *k* postavljenih literala gde je *k* parametar. Ova provera nam omogućava da na vreme “isečemo” prostor pretrage koji nije perspektivan zbog teorijskog konflikta, iako konflikt ne postoji za same iskazne promenljive odnosno klauze. U engleskoj literaturi postupak je poznat kao *early prunning* odnosno *eager notification*.
- Inkrementalni SAT rešavač (eng. on-line SAT solver) - osnovna ideja inkrementalnog SAT rešavača je da u toku rada možemo da dodamo novu klauzu. Kada je klauza dodata proverava se da li je ona konfliktna u odnosu na tekuću parcijalnu valuaciju. Ako jeste izvodi se povratni skok (eng. backjumping) na nivo odlučivanja gde ta dodata klauza nije bila konfliktna. Ova osobina se kombinuje sa inkrementalnim T-rešavačem. Kada dođe do teorijskog konflikta, T-rešavač proizvodi klauzu kojom se opisuje konflikt u teoriji. Ova klauza se dodaje u skup klauza i SAT rešavač nastavlja pretragu nakon povratnog skoka čime se dobija jako efikasna obrada teorijskog konflikta. Dodata klauza može odmah da se zaboravi, ali je ponekad bolje zadržati je da bi se izbegao takav konflikt kasnije u pretrazi. Čuvanje i zaboravljanje klauza je kompleksna stvar i tema sama za sebe, tako da nećemo ulaziti u moguće strategije.
- Teorijska propagacija - u prvo navedenoj osobini smo uveli pojam T-konzistentnosti i teorijskog konflikta. U tom slučaju mi pozadinsku teoriju koristimo tek nakon što je konflikt napravljen, pri čemu konflikt dolazi iz parcijalne valuacije, to jest proizveo ga je SAT rešavač koji ne zna ništa o pozadinskoj teoriji. Ideja teorijske propagacije je sledeća, umesto da čekamo na konflikt T-rešavač može u nekim slučajevima da detektuje da treba propagirati neki literal kao posledicu teorije. Iskazni literal koji odgovara atomu ili negaciji atoma koji je posledica teorije dodajemo u parcijalnu valuaciju. Pozadinska teorija jamči za postavku tog literala. Potrebno je napomenuti da je za neke teorije ovo moguće implementirati vrlo efikasno, međutim netrivijalan problem predstavlja analiza konflikata i povratni skok jer propagacija nije došla od SAT rešavača već od same teorije. Poslednja stvar koju treba znati je da se za neke teorije isplati *iscrpna teorijska propagacija* (eng. exhaustive theory propagation). Možda nije odmah jasno iz konteksta, ali teorijska propagacija se izvodi ako nije moguće primeniti jediničnu propagaciju iz samog SAT rešavača. Dakle, teorijsku propagaciju vršimo da izbegnemo da SAT rešavač slučajno (na osnovu neke heuristike) doda literal u parcijalnu valuaciju. Iscrpna teorijska propagacija se odnosi na to da umesto da propagiramo samo jedan literal, propagiramo sve literale koji su trenutna teorijska posledica.

Imajući prethodna poboljšanja u vidu, DPLL(T) je oznaka za arhitekturu gde primenjujemo prošireni DPLL algoritam pri čemu T označava da je postupak parametrizovan upotrebom teorijskog rešavača koji ćemo u nastavku obeležavati sa S_T. Standardni DPLL algoritam proširujemo sledećim pravilima:

- T-učenje (eng. T-learn) - M || F → M || F, C, kada se svaki atom iz C nalazi u F ili M i kada važi F $|=_T$ C (C je teorijska posledica uzimajući u obzir F)
- T-zaboravljanje (eng. T-forget) - M || F, C → M || F, kada je F $|=_T$ C
- T-povratni skok (eng. T-backjump) - umesto da se vraćamo za jedan nivo odlučivanja u nazad, odmah ćemo fomulisati naprednije pravilo koje nam omogućava povratni skok koji može da bude i po nekoliko nivoa. Pravilo glasi:
  $M/^dN$ || F, C → $M/'$ || F, C
  pri čemu je ispunjeno:
  - $M/^dN$ |= ~C
  - Postoji klauza C' ∨ l' takva da F, C $|=_T$ C' ∨ l' i M |= ~C'
  - l' je nedefinisan u M
  - l' ili ~l' se javlja u F ili $M/^dN$

Dakle, predstavljeni DPLL(T) koristi korake koje smo spominjali u okviru naprednih koncepata prilikom obrade originalnog DPLL algoritma. Ova tri pravila se u originalni algoritam uklapaju na sledeći način, kada dođe do teorijskog konflikta primenjujemo T-učenje da naučimo novu klauzu koja je teorijska posledica. Da se ne bi nakupilo previše ovakvih klauza primenjujemo T-zaboravljanje u skladu sa nekom heuristikom koja nam određuje koje klauze ćemo zaboraviti. Na kraju, prilikom konflikta ne vraćamo se jedan nivo unazad, već skačemo koliko nam stanje parcijalne valuacije i konfliktna klauza dozvoljavaju. DPLL algoritam i teorijski rešavač S_T komuniciraju na sledeći način:

- DPLL obaveštava S_T da je određeni literal dodat u parcijalnu valuaciju
- DPLL zahteva od S_T potvrdu da je tekuća parcijalna valuacija u skladu sa teorijom - frekventnost traženja ove potvrde zavisi od toga koliko je vremena potrebno rešavaču da izvrši ovu proveru. Uslov za S_T je da u slučaju teorijskog konflikta može da generiše objašnjenje za konflikt, to jest skup literala $\{l_1, \ldots, l_n\}$ koji se nalaze u parcijalnoj valuaciji takav da važi $|=_T$ ~l_1 ∨ … ∨ ~l_n. Ova klauza je zapravo objašnjenje konflikta. Idealno, što manje objašnjenje S_T uspe da generiše, to je bolje za efikasnost algoritma.
- DPLL zahteva od S_T da vrati listu literala koji su teorijska posledica trenutne parcijalne valuacije - ponovo imamo dodatan parametar kojim regulišemo vreme koje S_T može da utroši na ovaj zadatak. Ukoliko ne nađe nijedan odgovarajući literal S_T vraća praznu listu.
- DPLL zahteva od S_T da generiše objašnjenje za teorijsku propagaciju literala *l* - ova funkcionalnost je potrebna zbog povratka u pretrazi. Za razliku od slučaja u iskaznoj logici gde smo imali da je propagirani literal uvek propagiran od strane samog DPLL algoritma, sada imamo i teorijske propagacije.
- DPLL zahteva od S_T da poništi poslednjih *n* koraka u kojima su postavljane vrednosti literala.

Osim značajnih poboljšanja u smislu performansi, ovakva arhitektura može da kombinuje bilo koju implementaciju DPLL algoritma sa bilo kojim teorijskim rešavačem ukoliko i jedan i drugi mogu da ispoštuju gore naveden oblik komunikacije. Za čitaoca koji je zainteresovan za veći nivo detalja u odnosu na ovaj opis visokog nivoa, kao i dokaze korektnosti procedure, preporučujemo odličan pregledni rad.

Primena SMT rešavača

Instalacija Z3 rešavača na Linux sistemu

Z3 SMT rešavač razvija kompanija Microsoft za koju rade vodeći stručnjaci iz oblasti zbog čega je ovo jedan od najboljih dostupnih SMT rešavača. Kao takav biće korišćen u okviru ovog kursa. Osim Z3 rešavača, poznatiji rešavači su još i Yices, MathSAT 5 i Boolector.
Na linku se može preuzeti zip arhiva koja sadrži najnoviju stabilnu verziju rešavača. Arhivu otpakovati negde na fajl sistemu i pozicionirati se u taj direktorijum iz terminala. Nakon toga pokrenuti komande:

```
python scripts/mk_make.py
cd build
make -j
sudo make install
```

Ukoliko se desi da *make -j* poremeti normalan rad računara (slika na ekranu se zamrzne i sl.) pokrenuti samo *make* bez argumenta (argument *-j* služi za paralelnu kompilaciju). Nakon komande *sudo make install* važi sledeće:

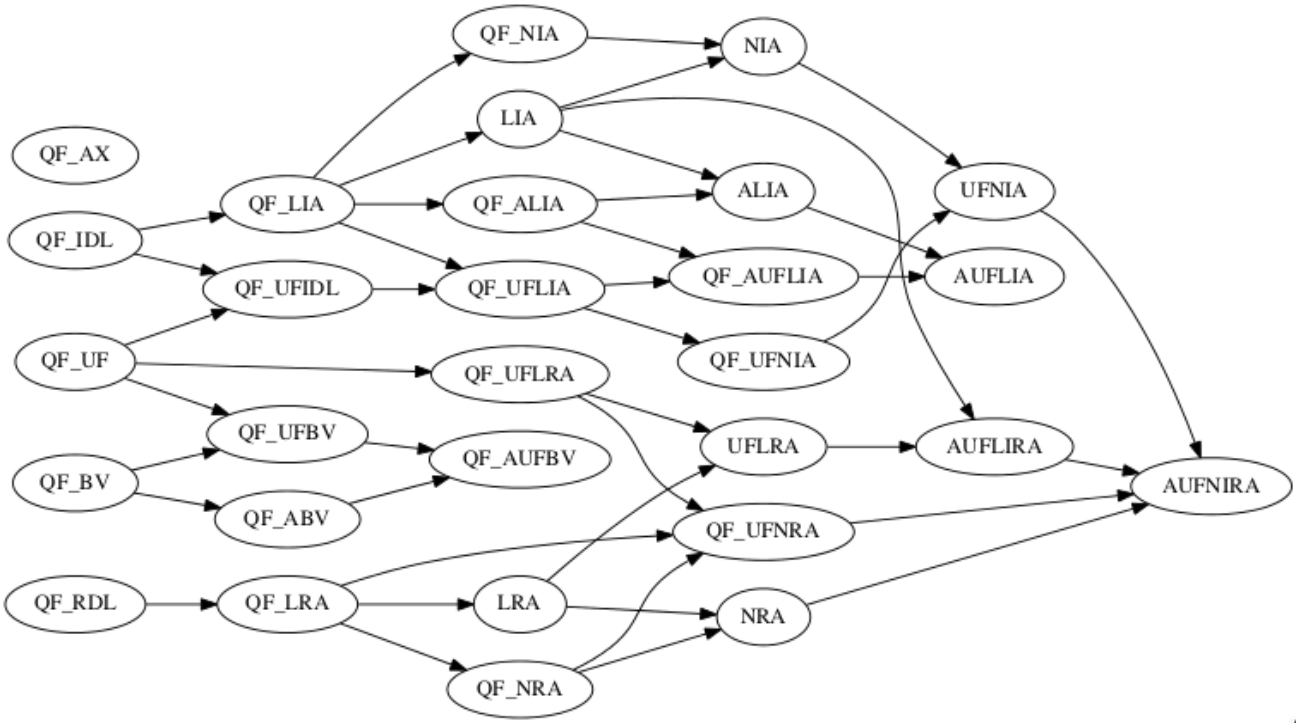
- U /usr/bin folderu nalazi se program *z3* koji pokrećete za fajlove u SMT-LIB 2 formatu:
  ```
  z3 fajl.smt2
  ```
- U /usr/include folderu se nalaze zaglavlja za pristup C, odnosno C++ API-ju Z3 rešavača (z3.h i z3++.h)
- U /usr/lib folderu se nalaze dinamičke biblioteke sa kojima se treba linkovati ukoliko program koji se kompajlira koristi C ili C++ API Z3 rešavača (biblioteka je libz3.so)

Kodiranje problema u SMT-LIB 2 formatu

SMT-LIB 2 format je tekstualni format za zapis formula neke teorije logike prvog reda. Ovo je standardizovan format što znači da se jednom zapisan problem može rešiti pomoću svih rešavača koji ovaj standard podržavaju. Samo postojanje standarda je prednost jer eliminiše potrebu da se korisnik rešavača upoznaje sa internim API-jem konkretnog rešavača koji je trenutno u upotrebi.
Na samom početku potrebno je izabrati teoriju i logiku kojoj formula pripada. To se radi komandom:
(set-logic *ime teorije*)
Imena teorija najčešće počinju sa QF_ što označava formule bez kvantifikatora (eng. quantifier free). Neke često podržane teorije su:

- QF_UF - EUF teorija
- QF_IDL - teorija celobrojnih razlika (eng. integer difference logic)
- QF_RDL - teorija realnih razlika (eng. real difference logic)
- QF_LRA - teorija linearne realne aritmetike
- QF_LIA - teorija celobrojne linearne aritmetike
- QF_BV - teorija bitvektora
- QF_AX - teorija nizova

Ovo su samo neke osnovne teorije, spisak standardizovanih naziva je značajno duži. Neke kompleksnije teorije se dobijaju kombinacijom osnovnih što se i vidi iz njihovog naziva. Na primer, QF_AUFLIA je teorija koja uključuje upotrebu nizova, neinterpretiranih funkcijskih simbola i linearne celobrojne aritmetike. Na sledećoj slici je prikazana većina postojećih teorija sa njihovim vezama:



Posmatrajući samo QF_LRA i QF_LIA može se uvideti da će biti potrebno definisati funkcijske konstante koje bi trebalo da imaju ograničenja da su u prvom slučaju realne, a u drugom celobrojne vrednosti. Ovo se rešava uvođenjem *sorti* odnosno tipova konstanti. Preciznije, samo rezonovanje se vrši u višesortnoj logici prvog reda. Pojam sorte odnosno tipa nećemo formalizovati, već ćemo se ograničiti na domen upotrebe koji bi trebalo da bude intuitivno jasan. Novi funkcijski simbol definišemo naredbom:

(declare-fun *simbol* (*tip prvog argumenta*, ...) *tip*)

dok se nova sorta definiše sa:

(declare-sort *simbol arnost*)

Prethodno navedene naredbe bismo mogli da iskombinujemo da definišemo *f(x)* gde *f* predstavlja neinterpretirani funkcijski simbol arnosti jedan:

(declare-logic QF_UF)
(declare-sort NoviTip 0)
(declare-fun f (NoviTip) NoviTip)

Formule čiju zadovoljivost treba ispitati zadajemo sa:

(assert *formula*)

pri čemu zadovoljivost odvojeno ispitujemo pozivom:

(check-sat)

nakon čega najčešće završavamo sa:

(exit)

Nekada nam nije dovoljno da ispitamo zadovoljivost neke formule već bismo želeli da znamo za koje vrednosti je formula zadovoljiva, to postižemo sa:

(get-value (*prva konstanta*, *druga konstanta*, ...))

Osnovna sort je Bool. Svaki predikat se evaluira u Bool. Druge značajne sorte su svakako Int i Real. Ove dve sorte su definisane u okviru QF_LIA i QF_LRA. Slede praktični primeri u kojima će se najbolje videti sintaksa zapisa i primene SMT rešavača na neke probleme.

Primer 1

Pokazati zadovoljivost sledeće formule korišćenjem SMT rešavača:

f(x) = y /\ g(y) = x /\ x != g(f(x))

Rešenje:

```
; Ovako izgleda komentar za SMT-LIB format
; Trebalo bi da bude jasno da je formula iz EUF teorije
(set-logic QF_UF)
(declare-sort S 0)
(declare-fun f (S) S)
(declare-fun g (S) S)
(declare-fun x () S)
(declare-fun y () S)

; Za nejednakost koristimo ključnu reč distinct
(assert (and (= (f x) y) (= (g y) x) (distinct x (g (f x)))))
(check-sat)
(exit)
```

Primer 2

SMT rešavačem rešiti sistem jednačina ako se zna da su *x* i *y* celi brojevi:

2x + 3y = 7

3x - 6y = 0

Rešenje:

```
; Ova formula potpada pod QF_LIA jer je uslov da su x i y celobrojni
(set-logic QF_LIA)
(declare-fun x () Int)
(declare-fun y () Int)
(assert (and (= (+ (* 2 x) (* 3 y)) 7) (= (- (* 3 x) (* 6 y)) 0)))
(check-sat)
(get-value (x y))
(exit)
```

Rešenje koje SMT daje je x=2, y=1.

Primer 3

SMT rešavačem rešiti sistem nejednačina ako su *x* i *y* realni brojevi:

2x + 3y > 1

3x - 2y < 0

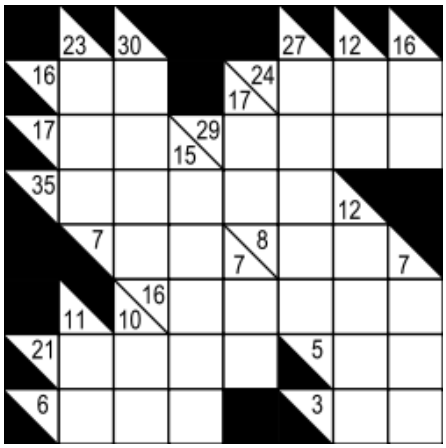
Rešenje:

```
; x i y su realni, koristimo QF_LRA
(set-logic QF_LRA)
(declare-fun x () Real)
(declare-fun y () Real)
(assert (and (< 1 (+ (* 2 x) (* 3 y))) (< (- (* 3 x) (* 2 y)) 0)))
(check-sat)
```

(get-value (x y))
(exit)
SMT rešavač vraća vrednosti x=0, y=2/3.

Primer 4

Neka je data sledeća Kakuro zagonetka:



Rešiti zagonetku uz pomoć SMT rešavača. Pomoć: u prazne kvadratiće se upisuju brojevi 1-9, brojevi u trouglovima označavaju zbirove koje treba postići popunjavanjem praznih kvadratića pri čemu u istom zbiru ne mogu učestvovati dva ista broja.
Rešenje:

```
; Sa Xij ćemo obeležavati prazan kvadratić u i-tom redu i j-toj koloni, ima 7 redova i kolona
(set-logic QF_LIA)

; Deklarišemo promenljive
(declare-fun x11 () Int)
(declare-fun x12 () Int)
(declare-fun x15 () Int)
(declare-fun x16 () Int)
(declare-fun x17 () Int)
(declare-fun x21 () Int)
(declare-fun x22 () Int)
(declare-fun x24 () Int)
(declare-fun x25 () Int)
(declare-fun x26 () Int)
(declare-fun x27 () Int)
(declare-fun x31 () Int)
(declare-fun x32 () Int)
(declare-fun x33 () Int)
(declare-fun x34 () Int)
(declare-fun x35 () Int)
(declare-fun x42 () Int)
(declare-fun x43 () Int)
(declare-fun x45 () Int)
(declare-fun x46 () Int)
```

```
(declare-fun x53 () Int)
(declare-fun x54 () Int)
(declare-fun x55 () Int)
(declare-fun x56 () Int)
(declare-fun x57 () Int)
(declare-fun x61 () Int)
(declare-fun x62 () Int)
(declare-fun x63 () Int)
(declare-fun x64 () Int)
(declare-fun x66 () Int)
(declare-fun x67 () Int)
(declare-fun x71 () Int)
(declare-fun x72 () Int)
(declare-fun x73 () Int)
(declare-fun x76 () Int)
(declare-fun x77 () Int)

; Formulu formiramo od 3 vrste uslova:
; 1) promenljive su između 1 i 9
(assert
 (and
  (<= 1 x11 9)
  (<= 1 x12 9)
  (<= 1 x15 9)
  (<= 1 x16 9)
  (<= 1 x17 9)
  (<= 1 x21 9)
  (<= 1 x22 9)
  (<= 1 x24 9)
  (<= 1 x25 9)
  (<= 1 x26 9)
  (<= 1 x27 9)
  (<= 1 x31 9)
  (<= 1 x32 9)
  (<= 1 x33 9)
  (<= 1 x34 9)
  (<= 1 x35 9)
  (<= 1 x42 9)
  (<= 1 x43 9)
  (<= 1 x45 9)
  (<= 1 x46 9)
  (<= 1 x53 9)
  (<= 1 x54 9)
  (<= 1 x55 9)
  (<= 1 x56 9)
  (<= 1 x57 9)
```

```
  (<= 1 x61 9)
  (<= 1 x62 9)
  (<= 1 x63 9)
  (<= 1 x64 9)
  (<= 1 x66 9)
  (<= 1 x67 9)
  (<= 1 x71 9)
  (<= 1 x72 9)
  (<= 1 x73 9)
  (<= 1 x76 9)
  (<= 1 x77 9)

; 2) promenljive koje učestvuju u istom zbiru su različite
; 3) zbir odgovarajućih promenljivih je jednak odgovarajućem broju iz zagonetke
; Za prvi red to znači da su x11 i x12 različiti i da je njihov zbir 16. Ostali uslovi se
; čitaju direktno iz zagonetke isto kao i ovaj uslov.
  (distinct x11 x12)
  (= (+ x11 x12) 16)
  (distinct x15 x16 x17)
  (= (+ x15 x16 x17) 24)
  (distinct x21 x22)
  (= (+ x21 x22) 17)
  (distinct x24 x25 x26 x27)
  (= (+ x24 x25 x26 x27) 29)
  (distinct x31 x32 x33 x34 x35)
  (= (+ x31 x32 x33 x34 x35) 35)
  (distinct x42 x43)
  (= (+ x42 x43) 7)
  (distinct x45 x46)
  (= (+ x45 x46) 8)
  (distinct x53 x54 x55 x56 x57)
  (= (+ x53 x54 x55 x56 x57) 16)
  (distinct x61 x62 x63 x64)
  (= (+ x61 x62 x63 x64) 21)
  (distinct x66 x67)
  (= (+ x66 x67) 5)
  (distinct x71 x72 x73)
  (= (+ x71 x72 x73) 6)
  (distinct x76 x77)
  (= (+ x76 x77) 3)

  (distinct x11 x21 x31)
  (= (+ x11 x21 x31) 23)
  (distinct x61 x71)
  (= (+ x61 x71) 11)
  (distinct x12 x22 x32 x42)
```

```
  (= (+ x12 x22 x32 x42) 30)
  (distinct x62 x72)
  (= (+ x62 x72) 10)
  (distinct x33 x43 x53 x63 x73)
  (= (+ x33 x43 x53 x63 x73) 15)
  (distinct x24 x34)
  (= (+ x24 x34) 17)
  (distinct x54 x64)
  (= (+ x54 x64) 7)
  (distinct x15 x25 x35 x45 x55)
  (= (+  x15 x25 x35 x45 x55) 27)
  (distinct x16 x26)
  (= (+ x16 x26) 12)
  (distinct x46 x56 x66 x76)
  (= (+  x46 x56 x66 x76) 12)
  (distinct x17 x27)
  (= (+ x17 x27) 16)
  (distinct x57 x67 x77)
  (= (+ x57 x67 x77) 7)
 )
)

; Provera zadovoljivosti i dohvatanje vrednosti promenljivih
(check-sat)
(get-value (x11 x12 x15 x16 x17))
(get-value (x21 x22 x24 x25 x26 x27))
(get-value (x31 x32 x33 x34 x35))
(get-value (x42 x43))
(get-value (x45 x46))
(get-value (x53 x54 x55 x56 x57))
(get-value (x61 x62 x63 x64))
(get-value (x66 x67))
(get-value (x71 x72 x73 x76 x77))
(exit)
```

Očekivani izlaz iz rešavača je:

```
((x11 9)
 (x12 7)
 (x15 8)
 (x16 7)
 (x17 9))
((x21 8)
 (x22 9)
 (x24 8)
 (x25 9)
 (x26 5)
 (x27 7))
```

```
((x31 6)
 (x32 8)
 (x33 5)
 (x34 9)
 (x35 7))
((x42 6)
 (x43 1))
((x45 2)
 (x46 6))
((x53 4)
 (x54 6)
 (x55 1)
 (x56 3)
 (x57 2))
((x61 8)
 (x62 9)
 (x63 3)
 (x64 1))
((x66 1)
 (x67 4))
((x71 3)
 (x72 1)
 (x73 2)
 (x76 2)
 (x77 1))
```

Primer 5

Golombov lenjir veličine *n* i reda *k* je skup od *k* celih brojeva $x_1 < x_2 < ... < x_k$ takvih da su sve razlike $x_j - x_i$ (za i < j) međusobno različite, kao i da je najveća među njima ($x_k - x_1$) jednaka *n*. Ispitati pomoću SMT rešavača da li postoji Golombov lenjir reda 6 i veličine 17. Napomena: s obzirom na to da pomeranje svih elemenata lenjira za *m* unapred ili unazad ne utiče na postojanje lenjira pretpostaviti $x_1 = 0$ i $x_6 = 17$.

Rešenje:

```
(set-logic QF_LIA)
(declare-fun x1 () Int)
(declare-fun x2 () Int)
(declare-fun x3 () Int)
(declare-fun x4 () Int)
(declare-fun x5 () Int)
(declare-fun x6 () Int)
(assert
  (and
    (distinct      (- x2 x1) (- x3 x1) (- x4 x1) (- x5 x1) (- x6 x1)
                   (- x3 x2) (- x4 x2) (- x5 x2) (- x6 x2)
                   (- x4 x3) (- x5 x3) (- x6 x3)
                   (- x5 x4) (- x6 x4)
```

```
                (- x6 x5))
   (= x1 0)
   (= x6 17)
   (< x1 x2 x3 x4 x5 x6)
  )
)
(check-sat)
(get-value (x1 x2 x3 x4 x5 x6))
(exit)
```

Vrednosti koje vraća SMT rešavač su 0, 5, 7, 13, 16, 17.

Primer 6

Za fragment koda:

```
int x, y;
...
if(x > y)
 max = x;
else
 max = y;
```

dokazati da je *max* jednak većoj od promenljivih *x* i *y*.

Rešenje:

```
; Koristimo teoriju bitvektora i pretpostavljamo širinu int-a 32 bita
(set-logic QF_BV)
(declare-fun x () (_ BitVec 32))
(declare-fun y () (_ BitVec 32))
(declare-fun m () (_ BitVec 32))

; Imamo početne naredbe čiju logiku kodiramo u formulu P, takođe
; imamo i postuslov koji treba da važi - nazovimo ga Q, kao i ranije
; pokazujemo ~(P => Q) to jest P /\ ~Q je nezadovoljiva
(assert (and
                ; P formula, ili je x > y i max=x ili...
                (or
                        (and (bvsgt x y) (= m x))
                        (and (not (bvsgt x y)) (= m y))
                )
                ; ~Q formula
                (not
                        (and (or (= m x) (= m y)) (bvsge m x) (bvsge m y))
                )
        )
)
(check-sat)
(exit)
```

Rešavač vraća *unsat*. Za kodiranje *if* uslova postoji skraćena naredba koju ćemo videti u narednim primerima. Bez obzira na njeno postojanje bitno je da čitalac shvati mehanizaciju kodiranja *if* uslova ručno.

Primer 7

Za sledeći fragment programskog koda:

```
 int i, n, max;
 int a[N];
 …
 max = a[0];
 i = 1;
 while(i < n)
 {
   if(a[i] > max)
    max = a[i];
   i++;
 }
```

dokazati da je *max* jednak najvećem elementu niza. Smatrati da je N >= n.

Rešenje:

```
; Petlje predstavljaju veliki problem za verifikaciju programa jer nema prirodnog načina
; da izrazimo šta je petlja zapravo. Zbog toga mi možemo da verifikujemo petlje samo
; za fiksan broj iteracija - moramo odabrati neku vrednost za n. Verifikaciju vršimo tako
; što razmotavamo petlju ručnu. Za i=1 promenljiva max ima neku vrednost, za i=2 ima
; neku drugu vrednost itd. Vrednosti promenljivih u različitim iteracijama kodiramo sa
; različitim promenljivama, vrednost max-a za i=1 sa m1, za i=2 sa m2 itd. Naravno ako
; se promenljive ne menjaju u telu petlje koristimo istu promenljivu - na primer za n.

; Logika mora da ima nizove i bitvektore, fiksiramo n=5
(set-logic QF_ABV)
(declare-fun i1 () (_ BitVec 32))
(declare-fun i2 () (_ BitVec 32))
(declare-fun i3 () (_ BitVec 32))
(declare-fun i4 () (_ BitVec 32))
(declare-fun i5 () (_ BitVec 32))
(declare-fun m1 () (_ BitVec 32))
(declare-fun m2 () (_ BitVec 32))
(declare-fun m3 () (_ BitVec 32))
(declare-fun m4 () (_ BitVec 32))
(declare-fun m5 () (_ BitVec 32))
(declare-fun a () (Array (_ BitVec 32) (_ BitVec 32)))

(assert
        (and
                ; Dodele pre petlje, (_ bvNekiBroj 32) označava dekadni broj NekiBroj
                ; zapisan kao 32bitni bitvektor u prvoj liniji broj 0, u drugoj broj 1
                (= m1 (select a (_ bv0 32)))
```

```
                (= i1 (_ bv1 32))

                ; Razmotano telo petlje, ite označava if-then-else
                (ite (bvsgt (select a i1) m1) (= m2 (select a i1)) (= m2 m1))
                (= i2 (bvadd i1 (_ bv1 32)))
                (ite (bvsgt (select a i2) m2) (= m3 (select a i2)) (= m3 m2))
                (= i3 (bvadd i2 (_ bv1 32)))
                (ite (bvsgt (select a i3) m3) (= m4 (select a i3)) (= m4 m3))
                (= i4 (bvadd i3 (_ bv1 32)))
                (ite (bvsgt (select a i4) m4) (= m5 (select a i4)) (= m5 m4))
                (= i5 (bvadd i4 (_ bv1 32)))

                ; Dodajemo postuslov, m5 je jednak nekom od elemenata niza pri čemu
                ; je veći ili jednak od svakog elementa niza za a[0], a[1], … ,a[4]. Opet
                ; uzimamo negaciju ovog uslova kao i u prethodnom primeru jer ispitujemo
                ; nezadovoljivost ~(P => Q) to jest P /\ ~Q.
                (not
                        (and
                                (or
                                        (= (select a (_ bv0 32)) m5)
                                        (= (select a (_ bv1 32)) m5)
                                        (= (select a (_ bv2 32)) m5)
                                        (= (select a (_ bv3 32)) m5)
                                        (= (select a (_ bv4 32)) m5)
                                )
                                (and
                                        (bvsge m5 (select a (_ bv0 32)))
                                        (bvsge m5 (select a (_ bv1 32)))
                                        (bvsge m5 (select a (_ bv2 32)))
                                        (bvsge m5 (select a (_ bv3 32)))
                                        (bvsge m5 (select a (_ bv4 32)))
                                )
                        )
                )
        )
)
(check-sat)
(exit)
```

Primer 8

Za dati fragment programskog koda:

```
double x, y, r;
...
if(x < y)
  r  = y - x;
else if(y < x)
```

```
    r  = x - y;
  else
    r = 1;
```

pokazati da je *r* različito od nula.

Rešenje:

Za ovaj zadatak će biti prikazana dva rešenja. Ispravnije rešenje zahteva korišćenje najsavremenijih tehnika koje nisu toliko rasprostranjene među današnjim SMT rešavačima. Radi se o teoriji realnih brojeva u pokretnom zarezu koja uzima u obzir odgovarajući IEEE standard koji propisuje osobine ovih brojeva. Alternativno rešenje se bazira na zanemarivanju stvarnih osobina brojeva u računaru i korišćenju teorije linearne realne aritmetike. Ovo rešenje će biti prikazano prvo, a zatim sledi rešenje koje uključuje teoriju realnih brojeva u pokretnom zarezu.

```
(set-logic QF_LRA)
(declare-fun x () Real)
(declare-fun y () Real)
(declare-fun r () Real)

(assert
        (and
                (ite (< x y)       (= r (- y x))
                                   (ite (< y x) (= r (- x y)) (= r 1))
                )
                (= r 0)
        )
)
(check-sat)
(exit)
```

Drugo rešenje korišćenjem QF_FP to jest formula bez kvantifikatora teorije aritmetike realnih brojeva u pokretnom zarezu:

```
; Logika je bez kvantifikatora i u pitanju su vrednosti u pokretnom zarezu
(set-logic QF_FP)

; Deklarišemo 64bitne konstante u pokretnom zarezu, eksponent ovakvog broja
; zauzima 11 bitova dok je mantisa 53 bita od čega se 52 čuva eksplicitno
(declare-fun x () Float64)
(declare-fun y () Float64)
(declare-fun r () Float64)

(assert
        (and
                ; Kada radimo operacije sa brojevima u pokretnom zarezu jedna od stvari
                ; o kojima moramo da vodimo računa je zaokruživanje - RNE je skraćenica
                ; za zaokruživanje na najbližu vrednost, u slučaju sredine zaokružiti na parnu.
                ; Zaokruživanje se ne koristi prilikom poređenja, dodatno ne smemo koristiti
                ; simbol = za jednakost već fp.eq. Konstante najlakše formiramo konverzijom:
                ; ((_ to_float BitovaEksponent BitovaMantisa) Zaokruživanje Vrednost)
                (ite (fp.lt x y)    (fp.eq r (fp.sub RNE y x))
```

```
                                        (ite (fp.lt y x)    (fp.eq r (fp.sub RNE x y))
                                                            (fp.eq r ((_ to_fp 11 53) RNE 1))))
                    (fp.eq r ((_ to_fp 11 53) RNE 0))
            )
)
(check-sat)
(exit)
```

Primer 9

Petao, Gavran i Kukavica se takmiče u pevanju. U žiriju se nalazi određeni broj sudija. Svaki sudija je glasao za jednog od takmičara. Detlić je bio zadužen za brojanje glasova. Ukupno je izbrojao 59 glasova. Po njegovom brojanju, 15 ih je glasalo za Petla ili Gavrana (u smislu da je zbir njihovih glasova toliki), 18 za Gavrana ili Kukavicu, a 20 za Kukavicu ili Petla. Detliću brojanje nije jača strana, tako da ni jedan od ova četiri broja ne mora biti tačan, ali se zna da ni u jednom slučaju nije pogrešio za više od 13. Koji su rezultati glasanja?
Zadatak rešiti SMT rešavačem. Takođe, pomoću SMT rešavača dokazati da je rešenje jedinstveno.
Rešenje:

```
; Direktno kodiramo uslove zadatka da bismo dobili rešenje
(set-logic QF_LIA)
(declare-fun ukupno () Int)
(declare-fun petao () Int)
(declare-fun gavran () Int)
(declare-fun kukavica () Int)

(assert
        (and
                (<= 46 ukupno 72)
                (<= 2 (+ petao gavran) 28)
                (<= 5 (+ kukavica gavran) 31)
                (<= 7 (+ kukavica petao) 33)
                (= ukupno (+ petao gavran kukavica))
        )
)
(check-sat)
(get-value (ukupno petao gavran kukavica))
(exit)
```

Dobijeno rešenje koje rešavač daje je:

```
sat
((ukupno 46)
 (petao 15)
 (gavran 13)
 (kukavica 18))
```

Proveravamo da li je rešenje jedinstveno tako što dodajemo različitosti promenljivih od ovih vrednosti u početno rešenje i dobijamo:

```
(set-logic QF_LIA)
(declare-fun ukupno () Int)
```

```
(declare-fun petao () Int)
(declare-fun gavran () Int)
(declare-fun kukavica () Int)

(assert
	(and
		(<= 46 ukupno 72)
		(<= 2 (+ petao gavran) 28)
		(<= 5 (+ kukavica gavran) 31)
		(<= 7 (+ kukavica petao) 33)
		(= ukupno (+ petao gavran kukavica))

		; Provera jedinstvenosti rešenja
		(distinct ukupno 46)
		(distinct petao 15)
		(distinct gavran 13)
		(distinct kukavica 18)
	)
)
(check-sat)
(exit)
```

Rešavač vraća *unsat* dakle rešenje je jedinstveno.

Primer 10

Odrediti pozicije 8 kraljica na šahovskoj tabli tako da se kraljice međusobno ne napadaju. Pomoć: šahovska tabla je dimenzija 8x8 to jest ima 64 polja; kraljica je figura koja se po šahovskoj tabli može kretati proizvoljan broj polja gore, dole, levo, desno i po dijagonalama.

Rešenje:

Prvo i verovatno jednostavnije rešenje do kojeg se može doći je da svako od polja šahovske table kodiramo binarnom promenljivom - ako je polje 1 tu se nalazi kraljica, dok ako je polje nula ne nalazi se kraljica. Potrebno je napraviti uslove takve da se u svakom redu nalazi bar jedna kraljica, u svakoj koloni se nalazi bar jedna kraljica, ni u jednom redu se nalazi proizvoljan par kraljica, ni u jednoj koloni se ne nalazi proizvoljan par kraljica i na kraju da nijedna dijagonala ne sadrži par kraljica. Ako su promenljive x_{ij} za polje table koje odgovara i-tom redu i j-toj koloni ove uslove nije teško zapisati, ali ih ima dosta (već smo prezentovali komplikovanija kodiranja problema - na primer igra sudoku). Ovakvo kodiranje je prirodno za SAT rešavače, ali ako koristimo SMT rešavače ne moramo koristiti binarne promenljive. Ideja je da iskoristimo ovu činjenicu da dobijemo jednostavnije kodiranje.

Svakom redu ćemo dodeliti promenljivu koja označava gde se kraljica nalazi u tom redu, obeležavaćemo promenljive sa x_i i imaćemo promenljive x_1, …, x_8. U svakom redu kraljica može da se nađe u jednoj od 8 kolona pa promenljive x_i uzimaju vrednosti iz opsega od 1 do 8. Ovakvim kodiranjem smo obezbedili da dve kraljice ne mogu da se nađu u istom redu. Uvodimo uslov da su vrednosti svih x_i međusobno različite čime obezbeđujemo da ne postoje dve kraljice u istoj koloni (vrednost promenljive x_i odgovara koloni u i-tom redu gde se nalazi kraljica). Za dijagonale uslov nije možda očigledan. Recimo da postavimo kraljice tako da važi $x_1 = 7$ i $x_4 = 4$. Crtanjem šahovske table može se videti da postoji dijagonala koja ih sadrži. Takođe može se videti da je apsolutna

razlika redova u kojima se one nalaze i kolona na kojima leže međusobno jednaka (iznosi 3). Baš ovaj uslov, samo negiran, ćemo koristiti da kodiramo da dve kraljice ne smeju biti na istoj dijagonali. Za svaki par redova njihova apsolutna razlika mora da se razlikuje od apsolutne razlike kolona.

```
(set-logic QF_LIA)
(declare-fun x1 () Int)
(declare-fun x2 () Int)
(declare-fun x3 () Int)
(declare-fun x4 () Int)
(declare-fun x5 () Int)
(declare-fun x6 () Int)
(declare-fun x7 () Int)
(declare-fun x8 () Int)

(assert
	(and
		(<= 1 x1 8)
		(<= 1 x2 8)
		(<= 1 x3 8)
		(<= 1 x4 8)
		(<= 1 x5 8)
		(<= 1 x6 8)
		(<= 1 x7 8)
		(<= 1 x8 8)

		(distinct x1 x2 x3 x4 x5 x6 x7 x8)

		(distinct (abs (- x1 x2)) 1)
		(distinct (abs (- x1 x3)) 2)
		(distinct (abs (- x1 x4)) 3)
		(distinct (abs (- x1 x5)) 4)
		(distinct (abs (- x1 x6)) 5)
		(distinct (abs (- x1 x7)) 6)
		(distinct (abs (- x1 x8)) 7)

		(distinct (abs (- x2 x3)) 1)
		(distinct (abs (- x2 x4)) 2)
		(distinct (abs (- x2 x5)) 3)
		(distinct (abs (- x2 x6)) 4)
		(distinct (abs (- x2 x7)) 5)
		(distinct (abs (- x2 x8)) 6)

		(distinct (abs (- x3 x4)) 1)
		(distinct (abs (- x3 x5)) 2)
		(distinct (abs (- x3 x6)) 3)
		(distinct (abs (- x3 x7)) 4)
		(distinct (abs (- x3 x8)) 5)
```

```
                (distinct (abs (- x4 x5)) 1)
                (distinct (abs (- x4 x6)) 2)
                (distinct (abs (- x4 x7)) 3)
                (distinct (abs (- x4 x8)) 4)

                (distinct (abs (- x5 x6)) 1)
                (distinct (abs (- x5 x7)) 2)
                (distinct (abs (- x5 x8)) 3)

                (distinct (abs (- x6 x7)) 1)
                (distinct (abs (- x6 x8)) 2)

                (distinct (abs (- x7 x8)) 1)
        )
)
(check-sat)
(get-value (x1 x2 x3 x4 x5 x6 x7 x8))
(exit)
```

Rešavač vraća rezultat (za vežbu nacrtati kraljice na šahovskoj tabli kao proveru):

```
sat
((x1 5)
 (x2 7)
 (x3 1)
 (x4 3)
 (x5 8)
 (x6 6)
 (x7 4)
 (x8 2))
```

Integracija SMT rešavača u aplikacije

SMT-LIB format je zgodan za jednostavan i svima čitljiv zapis problema. Korisnik čak ne mora da bude baš programer da bi iskoristio rešavač (naravno tehnička obučenost jeste potrebna). Međutim ovakav zapis problema ima dve velike mane:

1. Neki problemi zahtevaju automatsko generisanje uslova zbog velikog broja uslova i ne mogu se ručno zapisati u razumnom vremenu
2. SMT-LIB format nije zgodan za aplikacije kojima je SMT rešavač samo alat koji koriste za ispunjenje nekog drugog cilja (npr. automatsko generisanje rasporeda časova)

Moderni SMT rešavači najčešće dolaze sa dinamičkom bibliotekom i fajlovima zaglavlja za programski jezik C koji se uobičajeno koristi za implementaciju rešavača zbog efikasnosti. Dodatno, neki rešavači pružaju interfejs i za druge programske jezike i okruženja kao što su C++, Java, Python i Matlab. Jedan od ovakvih rešavača je i već pomenuti Z3 rešavač čiji ćemo C++ API iskoristiti da isprogramiramo malu aplikaciju koja će nam pomoći da rasporedimo virtuelne mašine na servere u oblaku (eng. cloud) i tako pomognemo jednoj kompaniji da ima što profitabilnije poslovanje.

Primer korišćenja SMT rešavača u okviru praktične aplikacije

Neka je dato *m* virutalnih mašina pri čemu su d_0, …, d_{m-1} količine prostora na disku koji ove virtualne mašine zauzimaju. Pored njih imamo i *n* servera koji imaju kapacitete diskova c_0, …, c_{n-1} i dnevnu cenu održavanja p_0, …, p_{n-1} koja mora da se plati ako je server u upotrebi. Rasporediti ove virtualne mašine na servere tako da što manje servera bude iskorišćeno i da dnevna cena koju kompanija plaća za servere bude što manja.

Iako tekst možda ne deluje komplikovano, ovaj problem je prilično težak za rešiti. Prvo, može da se desi da nema rešenja, na primer imamo dva servera kapaciteta 100GB i jednu virtualnu mašinu koja zauzima 120GB. Iako je ukupan prostor 200GB mi nemamo više od 100GB celog prostora tako da nema rešenja. Druga stvar o kojoj treba voditi računa je da je u pitanju problem sa više ciljeva - što manji broj servera i što manja dnevna cena. Posmatrajmo još jedan jednostavan primer, neka su date virtualne mašine koje zahtevaju 100GB, 50GB i 15GB prostora i neka imamo servera kapaciteta 100GB, 75GB i 175GB čija je dnevna cena 10$, 5$ i 20$ respektivno. Može se primetiti da ako stavimo sve virtualne mašine na poslednji server imamo da smo iskoristili samo jedan server, što je odlično, međutim cena nam je 20$. S druge strane ako stavimo prvu mašinu na prvi server, a zatim drugu i treću mašinu na drugi server dobijamo da smo iskoristili dva servera, međutim sada nam je cena niža i iznosi 15$.

Pretpostavimo za trenutak da problem ima bar neko rešenje. Ako nema smatraćemo da ćemo SMT rešavačem uspeti da dokažemo nezadovoljivost. U prethodnom primeru navedena dva rešenja minimizuju jedan od ciljeva i u tom smislu su optimalna. Ni za jedno od prethodnih rešenja nismo mogli da poboljšamo jedan cilj, bez da pokvarimo vrednost za drugi cilj. Kada rešenja imaju ovu osobinu kažemo da su Pareto optimalna ili Pareto efikasna. Ova imena potiču od ekonomiste Vilijama Pareta. Ono što je za nas zanimljivije je da ćemo iskoristiti SMT rešavač da izlistamo sva Pareto optimalna rešenja za dati ulaz i tako omogućiti kompaniji da odabere za sebe rešenje koje pruža najbolji balans za njihovu strategiju poslovanja.

Koristićemo sledeće kodiranje binarna promenljiva x_{ij} će da označava da smo i-tu virtualnu mašinu postavili na j-ti server. Pored toga uvodimo i binarne promenljive y_j koje označavaju da li je server u upotrebi. Ukupan broj promenljivih je m * n + n. Treba da zapišemo sledeće uslove:

- Virtualna mašina se može naći na samo jednom od servera
  - Najviše jedna promenljiva x_{i*} je 1
- Ukoliko se bar jedna virtualna mašina nalazi na serveru znači da je on u upotrebi
  - $y_j = 1$ ako je bar jedan $x_{*j} = 1$
- Zbir resursa virtualnih mašina dodeljenih serveru mora da se uklopi u resurse servera
  - Zbir utrošenog prostora mora da bude manji od kapaciteta servera: $y_j * c_j >= \sum_{i}^{m} x_{ij} * d_i$
- Uslove koji su ciljevi:
  - Zbir y_j promenljivih treba da bude što manji (broj servera u upotrebi)
  - Zbir $y_j * p_j$ treba da bude što manji (ukupna cena servera u upotrebi)

Kao ulaz aplikacija sa standardnog ulaza čita podatke u sledećem redosledu:

m
d_0 d_1 … d_{m-1}
n
c_0 c_1 … c_{n-1}
p_0 p_1 … p_{n-1}

i kao izlaz ispisuje sve Pareto optimalne modele ukoliko rešenje uopšte postoji. U nastavku sledi implementacija rešenja korišćenjem Z3 C++ API-ja:

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <iterator>
#include <string>
#include <map>
#include <set>

#include <z3++.h>

static void printModel(const z3::model &model, const std::vector<unsigned> &serversPrice);
static void virtualMachinesPlacement(std::istream &inStream);

int main()
{
   std::cout << "Unesite parametre u sledecem redosledu:\n"
                "m           \n"
                "d0 d1 ... dm-1\n"
                "n           \n"
                "c0 c1 ... cn-1\n"
                "p0 p1 ... pn-1\n";
   try {
       virtualMachinesPlacement(std::cin);
   } catch (const z3::exception &e) {
       std::cerr << "Z3 exception: " << e.msg() << "\n";
       std::exit(EXIT_FAILURE);
   }

   return 0;
}

static void virtualMachinesPlacement(std::istream &inStream)
{
   /* Ucitavamo parametre problema sa ulaznog stream-a */
   unsigned n, m;
   std::vector<unsigned> vmsSpaceNeeded;
   std::vector<unsigned> serversCapacity;
   std::vector<unsigned> serversPrice;
   inStream >> m;
   vmsSpaceNeeded.reserve(m);
   std::copy_n(std::istream_iterator<unsigned>{inStream},
               m,
               std::back_inserter(vmsSpaceNeeded));
   inStream >> n;
   serversCapacity.reserve(n);
   serversPrice.reserve(n);
   std::copy_n(std::istream_iterator<unsigned>{inStream},
               n,
               std::back_inserter(serversCapacity));
   std::copy_n(std::istream_iterator<unsigned>{inStream},
               n,
               std::back_inserter(serversPrice));
```

```
/* Kreiramo z3 kontekst; sve promenljive, uslove i resavace cemo vezati za
 * ovaj kontekst. Takodje, kreiramo objekat za konfigurisanje opcija Z3 resavaca
 * da bismo mogli da nadjemo sva Pareto optimalna resenja.
 */
z3::context ctx;
z3::params params{ctx};
params.set("priority", ctx.str_symbol("pareto"));

/* Kreiramo optimizator u koji cemo dodavati uslove. Obicno se koristi
 * z3::solver umesto z3::optimize objekta medjutim nas problem je da
 * optimizujemo neke vrednosti sto je kompleksnije od ispitivanja za-
 * dovoljivosti. Resavac z3::solver bi nam vratio neki model, ali nema
 * garancija da bi vratio optimalni (ukoliko model uopste postoji). Pored
 * ovoga prosledjujemo opcije optimizatoru (Pareto optimalna resenja).
 */
z3::optimize opt{ctx};
opt.set(params);

/* Kreiramo vektore x i y za promenljive koje cemo koristiti u uslovima */
z3::expr_vector x{ctx};
z3::expr_vector y{ctx};
for (unsigned i = 0; i < m; ++i)
{
    for (unsigned j = 0; j < n; ++j)
    {
        std::string name = "x_";
        name = name + std::to_string(i) + "_" + std::to_string(j);
        x.push_back(ctx.bool_const(name.c_str()));
    }
}
for (unsigned j = 0; j < n; ++j)
{
    y.push_back(ctx.bool_const(("y_" + std::to_string(j)).c_str()));
}

/* Najvise jedna promenljiva Xi* je jednaka jedinici (i-ti VM radi na najvise
 * jednom serveru)
 */
z3::expr zero = ctx.int_val(0);
z3::expr one = ctx.int_val(1);
for (unsigned i = 0; i < m; ++i)
{
    z3::expr_vector ithRow{ctx};
    for (unsigned j = 0; j < n; ++j)
    {
        ithRow.push_back(z3::ite(x[i * n + j], one, zero));
    }
    opt.add(z3::sum(ithRow) == one);
}

/*
 * Oznacavamo da je j-ti server u upotrebi ako j-ta kolona ima bar jednu jedinicu,
 * to jest bar jedan VM je rasporedjen na j-ti server.
```

```
 */
for (unsigned j = 0; j < n; ++j)
{
    z3::expr_vector jthCol{ctx};
    for (unsigned i = 0; i < m; ++i)
    {
        jthCol.push_back(x[i * n + j]);
    }
    opt.add(y[j] == z3::atleast(jthCol, 1U));
}

/*
 * Kodiramo uslove kapaciteta, suma zauzeca po svim VM-ovima na j-tom serveru mora
 * da bude manja od kapaciteta j-tog servera. Moramo da pazimo da su x i y Bool
 * sorta sto znaci da ih moramo konvertovati u Int sortu i to vrednosti 0 ili 1.
 */
for (unsigned j = 0; j < n; ++j)
{
    z3::expr sum = ctx.int_val(0);
    for (unsigned i = 0; i < m; ++i)
    {
        sum = sum + z3::ite(x[i*n + j], one, zero) * ctx.int_val(vmsSpaceNeeded[i]);
    }
    opt.add(sum <= z3::ite(y[j], one, zero) * ctx.int_val(serversCapacity[j]));
}

/*
 * Dodajemo optimizacione ciljeve da je ukupan broj servera sto manji i
 * da je ukupna cena sto manja.
 */
z3::expr nServers = ctx.int_val(0);
z3::expr price = ctx.int_val(0);
for (unsigned j = 0; j < n; ++j)
{
    z3::expr y_j = z3::ite(y[j], one, zero);
    nServers = nServers + y_j;
    price = price + y_j * ctx.int_val(serversPrice[j]);
}
opt.minimize(nServers);
opt.minimize(price);

/* Stampamo sve modele ako ih ima */
z3::check_result result;
unsigned modelsCnt = 0U;
while (z3::sat == (result = opt.check()))
{
    std::cout << "Model " << modelsCnt << ":\n";
    printModel(opt.get_model(), serversPrice);
    ++modelsCnt;
}

/* Ako nismo nasli nijedan model ispisujemo poruku o tome */
if (!modelsCnt)
{
```

```
        std::cout << "Raspodela virtualnih masina na servere "
                     "nije pronadjena za date parametre.\n";
    }
}


static void printModel(const z3::model &model, const std::vector<unsigned> &serversPrice)
{
    /* Parsiramo model da vidimo koji VM je na kom serveru */
    std::multimap<int, int> vmOnSrvMap;
    std::set<int> serversUsed;
    int vmIdx, srvIdx;
    for (unsigned i = 0; i < model.size(); ++i)
    {
        /* Znamo da imamo samo konstante cija je sorta Bool */
        z3::func_decl constant = model[i];
        std::string constantName = constant.name().str();
        if (constantName[0] == 'x' &&
            Z3_L_TRUE == model.get_const_interp(constant).bool_value())
        {
            /* Dohvatamo indekse VM-a i servera i dodajemo u multi mapu i u skup */
            std::sscanf(constantName.c_str(), "x_%d_%d", &vmIdx, &srvIdx);
            serversUsed.insert(srvIdx);
            vmOnSrvMap.insert(std::make_pair(srvIdx, vmIdx));
        }
    }

    /* Za sve servere ispisujemo koji su VM-ovi na njima i racunamo ukupnu cenu */
    std::cout << "\tVirtualne masine su dodeljene sledecim serverima:\n";
    unsigned totalPrice = 0U;
    for (const int server : serversUsed)
    {
        /* Uvecavamo cenu */
        totalPrice += serversPrice[server];

        /*
         * Dohvatamo par iteratora izmedju kojih se nalaze svi elementi
         * ciji je kljuc 'server'. Kada se dereferencira sam iterator
         * dobije se par int-ova od kojih prvi ima vrednost 'server',
         * a drugi odgovara indeksu virtualne masine pridruzene tom serveru.
         */
        std::cout << "\t\tServer " << server << ": ";
        auto itPair = vmOnSrvMap.equal_range(server);
        while (itPair.first != itPair.second)
        {
            std::cout << "(VM " << itPair.first->second << ") ";
            ++itPair.first;
        }
        std:: cout << "\n";
    }

    std::cout << "\tUkupan broj servera u upotrebi: " << serversUsed.size() << '\n';
    std::cout << "\tUkupna dnevna cena servera: " << totalPrice << "$\n";
}
```

Napisani program za ulaz (3 virtualne mašine, 3 servera):
3
100 50 15
3
100 75 200
10 5 20
daje dva Pareto optimalna rešenja, ispis u terminalu je:
Model 0:
    Virtualne masine su dodeljene sledecim serverima:
        Server 2: (VM 2) (VM 1) (VM 0)
    Ukupan broj servera u upotrebi: 1
    Ukupna dnevna cena servera: 20$
Model 1:
    Virtualne masine su dodeljene sledecim serverima:
        Server 0: (VM 0)
        Server 1: (VM 1) (VM 2)
    Ukupan broj servera u upotrebi: 2
    Ukupna dnevna cena servera: 15$
S druge strane za ulaz (jedna virtualna mašina, 2 servera nedovoljnog kapaciteta):
1
150
2
100 100
10 10
ispisuje da model ne postoji:
Raspodela virtualnih masina na servere nije pronadjena za date parametre.
Ovim primerom završavamo gradivo kursa.