

UNIVERSITATEA "ALEXANDRU-IOAN CUZA" DIN IAȘI

FACULTATEA DE INFORMATICĂ



LUCRARE DE LICENȚĂ

Verificarea algoritmului DPLL in F^*

propusă de

Alexandru Donica

Sesiunea: iunie/iulie, 2023

Coordonator științific

Conf. Dr. Ștefan Ciobâcă

UNIVERSITATEA "ALEXANDRU-IOAN CUZA" DIN IAȘI

FACULTATEA DE INFORMATICĂ

Verificarea algoritmului DPLL în F^*

Alexandru Donica

Sesiunea: iunie/iulie, 2023

Coordonator științific

Conf. Dr. Ștefan Ciobâcă

Cuprins

Motivație	2
Introducere	3
1 Algoritmul DPLL (Davis-Putnam-Logemann-Loveland)	5
2 Detalii de implementare	8
2.1 Despre FStar	8
2.2 Completitudinea, corectitudinea și terminarea programului	8
2.2.1 Completitudinea programului	8
2.2.2 Corectitudinea programului	9
2.2.3 Terminarea programului	10
2.3 Structurile de date folosite	11
2.4 Modulele ce alcătuiesc programul	14
2.5 Metrici orientative	19
2.6 Posibile optimizări	20
3 Pași necesari pentru a reproduce	21
3.1 Programe si resurse necesare	21
3.2 Executarea solver-ului SAT	23
Concluzii	24
Bibliografie	25

Motivație

Rolul unui 'SAT solver' este de a rezolva problema satisfiabilității booleene, făcându-se și în ziua de azi cercetări care au scopul de a îmbunătăți algoritmi existenți. Acest 'SAT solver' găsește o soluție pentru o formulă dată în cazul în care formula este satisfiabilă, în caz contrar formula este nesatisfiabilă.

Corectitudinea oricărui rezultat al unei formule satisfiabile poate fi verificată folosind metode simple. Însă un 'SAT solver' complex creat pentru a procesa formule de dimensiuni din ce în ce mai mari sau pentru a avea o viteză de rezolvare a problemei mai rapidă decât alți 'SAT solveri', poate conține erori de programare ce ar produce rezultate false, cum ar fi, în urma procesării unei formule nesatisfiabile acesta să enunțe că este satisfiabilă, sau invers.

De aceea este importantă crearea unor programe care verifică corectitudinea acestor 'SAT solveri' și am creat un 'SAT solver' verificat formal folosind limbajul de programare F* (FStar).

Introducere

Problema satisfiabilității booleene (SAT) se regăsește în multe ramuri ale domeniului informaticii, precum în verificarea de software și hardware, în domeniul criptografiei și securității sau optimizarea și analiza altor programe.

'SAT' este considerată prima problema ce a fost demonstrată a fi NP-completă ¹, motiv pentru care este importantă dezvoltarea unor programe care să poată găsi o soluție cât mai rapid, eficient și sigur.

În ziua de astăzi exista și încă se dezvoltă și optimizează 'SAT solveri' eficienți, cum ar fi cei care au participat la concursul "SAT 2022": Mallob, sau Paracooba.

Algoritmul DPLL reprezintă una din primele optimizări eficiente folosite în rezolvarea problemei SAT. Este considerat a fi un punct de plecare fundamental pentru dezvoltarea unui 'SAT solver' eficient, stând la baza multor solveri moderni.

Însă chiar și o implementare a unui algoritm demonstrat a fi eficient și complet poate produce erori sau rezultate greșite, iar unul din motivele pentru care apar aceste erori este eroarea umana a programatorului, a cărei frecvență poate crește împreună cu complexitatea algoritmului ce trebuie implementat. Pentru că testarea unui 'SAT solver' nu poate garanta absența erorilor, s-au conceput programe care să verifice că implementarea în sine a algoritmului este lipsită de erori și matematic corectă.

Limbajul de programare F* (FStar) pune la dispoziție un astfel de sistem care în momentul compilării execută o verificare formală a programului în funcție de specificațiile introduse de programator. Acest sistem ajută la depistarea bug-urilor, prevenirea erorilor, și oferă o siguranță mai mare că programul este corect implementat și nu va prezenta rezultate false sau greșite.

Astfel, am ales să implementez algoritmul DPLL folosind limbajul de programare FStar, pentru a evidenția cum arată parcursul creării și verificării unui 'SAT solver' în acest limbaj și cum se poate extinde din punct de vedere al eficientizării.

¹https://en.wikipedia.org/wiki/Cook%E2%80%93Levin_theorem

Această lucrare va prezenta în continuare o implementare a algoritmului DPLL folosind limbajul FStar. Vor fi prezentate detalii despre algoritmul DPLL, despre limbajul de programare folosit, detalii de implementare cu exemple de cod și motivele pentru care 'SAT solver-ul' rezultat este complet, corect și produce mereu un rezultat.

Capitolul 1

Algoritmul DPLL

(Davis-Putnam-Logemann-Loveland)

Algoritmul DPLL creat pentru rezolvarea problemei SAT este considerat o optimizare a metodei de backtracking și stă la baza creării a multor 'SAT solveri' moderni.

În cadrul acestui algoritm a fost demonstrată proprietatea că pentru orice formulă care se poate scrie în formă normală conjunctivă (CNF), se va returna o soluție validă dacă aceasta există, și se va specifica mereu că nu există dacă formula este într-adevăr nesatisfiabilă.

Optimizarea cea mai importantă pe care acest algoritm o aduce metodei backtracking este 'propagarea clauzelor unit', unde o 'clauză unit' este o clauză ce nu conține nici un literal cu valoarea *True* și care conține un singur literal fără valoare aleasă.

Astfel, la fiecare pas recursiv, se caută fiecare clauză 'unit' și se asignează o valoare variabilei rămase, astfel încât literalul din clauza respectivă să aibă valoarea *True*. Motivul pentru care această metodă funcționează este că dacă s-ar asigna valoarea *False* unui astfel de literal, valoarea de adevăr a clauzei respective ar fi *False*, deci soluția ar fi invalidă.

Urmează pseudo-codul metodei recursive '*dpll*', formulată după algoritmul DPLL modificat să corespundă implementării acestui proiect.

Input: formula : $f \{ \text{length } f \geq 1 \}$;

truth_assignment : $\tau \{ \text{length } \tau \leq \text{length } f \wedge \text{is_partial_solution } f \tau \}$

Output: result : $r \{ (r \text{ is (Sat } \tau) \implies \text{calculate_formula_value } f \tau = \text{True})$
 $\wedge (r \text{ is Unsat} \implies \forall \text{ truth_assignment : } t .$
 $\text{calculate_formula_value } f t = \text{false}) \}$

if $\text{length } \tau = \text{length } (\text{variables_of } f)$ **then**
 | **return** Sat τ
end

else
 $\text{new_var} \leftarrow \text{get_unassigned_variable } f \tau$
 $\tau_2 \leftarrow \tau[\text{new_var} : \text{False}]$
 $\tau_2, \text{new_literals} \leftarrow \text{unit_clause_propagation } f \tau_2$
 $\text{clauses_with_new_literals} \leftarrow \text{get_clauses_that_contains_min_1_literal } f \text{new_literals}$
 if $\exists c : \text{clause } c \in \text{clauses_with_new_literals} \wedge \text{is_clause_false } c \tau_2$ **then**
 | $\tau_2 \leftarrow \tau[\text{new_var} : \text{True}]$
 ...aceeiasi pasi ca pana acum, dar cu noul τ_2 ,si returneaza NotSat în
 block-ul if-then corespondent...
 end
 else
 $\text{res} \leftarrow \text{dpll_recursiv } f \tau_2$
 if res is NotSat **then**
 | $\tau_2 \leftarrow \tau[\text{new_var} : \text{True}]$
 ...aceeiasi pasi ca pana acum, dar cu noul τ_2 ,si returneaza NotSat
 în block-ul if-then corespondent...
 end
 else
 | **return** res
 end
 end
end

Algorithm 1: dpll-recursiv

Exemplu: formulă ce conține 5 clauze, numerotate de la c_1 la c_5 și 6 variabile numerotate de la x_1 la x_6

$$(\neg x_1 \vee x_3 \vee x_4) \wedge (\neg x_2 \vee \neg x_3 \vee x_5) \wedge (x_1 \vee \neg x_2) \wedge (x_4 \vee \neg x_5) \wedge (x_1 \vee \neg x_4 \vee \neg x_6)$$

Programul ar începe căutarea unei soluții asignând unei variabile noi o valoare oarecare, mai exact, variabilei x_1 valoarea *False*. Urmează etapa de *unit propagation*, în care se găsește clauza c_3 a fi clauză unitară, și se asignează variabilei x_2 valoarea *False*. Apoi, recursiv, se ia logica de la capăt cu datele noi obținute. Se va găsi primul literal neassignat, x_3 , și i se va acorda valoarea *False*. În etapa de propagare a clauzelor unitare se găsește clauza unitară c_1 , ce conține 2 literali cu valoare falsă și variabila x_4 , care va primi valoarea *True*. Tot în etapa de propagare a clauzelor unitare, se găsește acum clauza c_5 , unde singurul literal rămas este $\neg x_6$, și deci variabila x_6 va primi valoarea *False*. Astfel toate clauzele au valoarea *True*, iar variabila x_5 poate avea orice valoare de adevăr. Deci s-a găsit o soluție validă și formula este satisfiabilă.

$$x_1 = False; x_2 = False; x_3 = False; x_4 = True; x_5 = orice; x_6 = False$$

Capitolul 2

Detalii de implementare

2.1 Despre FStar

Limbajul F* este un limbaj de programare orientat pe demonstrații ce poate fi folosit pentru uz general. Printre aspectele acestui limbaj se numără și faptul că prezintă suport pentru programare funcțională. De asemenea, este proiectat să execute o verificare formală a codului, astfel orice programator poate să adauge funcțiilor specificații sub formă de pre-condiții, post-condiții, invariante, pentru a putea asigura completitudinea codului.

2.2 Completitudinea, corectitudinea și terminarea programului

Cele mai importante proprietăți pe care le respectă acest 'SAT solver' sunt completitudinea algoritmului, corectitudinea implementării și siguranța că programul se va termina și va produce un rezultat.

2.2.1 Completitudinea programului

Este dovedită completitudinea algoritmului DPLL pentru o formulă în formă normală conjunctivă (CNF), așa cum este el descris în Capitolul 1.

Singura optimizare a algoritmului DPLL care a fost implementată se leagă de verificarea existenței unei clauze false la fiecare pas recursiv după etapa de '*unit propagation*'.

Mai exact, la fiecare pas există o listă de literali, de lungime cel puțin 1, care au fost adăugați la soluție, dar acești literali noi nu schimbă valoarea clauzelor care nu conțin nici unul din literalii din listă.

Astfel, această optimizare se folosește de o matrice de apariție care, pentru fiecare literal din formulă, reține câte o listă cu toate clauzele în care apare respectivul literal. Se intersectează clauzele în care apar toți acești literali asignați la pasul curent, iar căutarea unei clauze false se face doar peste această mulțime, în loc de întreaga formulă.

2.2.2 Corectitudinea programului

Pentru verificarea specificațiilor introduse de programator, se folosește un SMT solver numit Z3¹. Cu toate acestea, unele condiții pot fi prea dificile de demonstrat într-un punct anume al programului, unul din motivele pentru care limbajul F* oferă la dispoziția programatorilor instrucțiuni ce simplifică misiunea de a demonstra și verifică proprietăți și condiții scrise în program. Recurente în cadrul acestui proiect sunt structurile *'Lemma'* și instrucțiunile *'assert'* care ajută la demonstrarea unor proprietăți între diferite tipuri de date ce respectă anumite condiții, și ajută 'SMT solver-ul' spunându-i să demonstreze întâi proprietăți intermediare.

Exemplu *'lemma'*:

```
let rec lemma_values_from_clause_same_for_all_supra_set_of_t
(t : truth_assignment) (c : clause) : Lemma
(requires are_clause_vars_in_assignment t c)
(ensures
  (forall (other_t : truth_assignment
    {forall (v : variable_info {L.mem v t}).
      (L.mem v other_t)}}).
    (L.mem true (get_values_from_clause t c)
      = L.mem true (get_values_from_clause other_t c))))
)
=
if length c = 1
then ()
else lemma_values_from_clause_same_for_all_supra_set_of_t t (L.tl c)
```

¹https://en.wikipedia.org/wiki/Z3_Theorem_Prover

Această *lemma* ajută la demonstrarea faptului că pentru orice clauză c , ai cărei literali au o valoare asignată în mulțimea t , atunci valoarea de adevăr a clauzei c în corespondență cu mulțimea t va fi egală cu valoarea de adevăr a lui c în corespondență cu orice mulțime care este supra-set pentru t .

Exemplu instrucțiune 'assert':

```
assert (L.noRepeats ress);
```

Această simplă instrucțiune ajută a arăta faptul că la momentul verificării acestei instrucțiuni, lista 'ress' nu conține elemente duplicate.

Exemple asemănătoare se vor găsi și în secțiunile ce urmează.

După verificarea unui fișier '.fst' se va extrage, în fișiere tipice limbajului Ocaml, codul ce este folosit la formarea rezultatului, și este eliminat codul ce ajută la specificații și demonstrarea unor proprietăți. Cu cât validările devin mai complexe, cu atât se poate observa diferența mare de linii de cod dintre fișierele '.fst' și cele '.ml'.

De exemplu, modulul 'DpllPropagation' conține multe instrucțiuni ce ajută la verificare, în total ajungând la ≈ 600 de linii fișierul, însă corespondentul modul Ocaml extras din cel FStar, are ≈ 100 linii, însemnând că s-au eliminat cele ≈ 500 linii care contribuiau la verificarea formală a modului.

2.2.3 Terminarea programului

Terminarea programului și producerea unui rezultat trebuie să fie o certitudine, indiferent de valoarea rezultatului. Limbajul FStar poate verifica, dacă este specificat acest lucru, dacă o funcție/un program se va termina și va produce un rezultat indiferent de parametrii primiți. De asemenea, poate verifica dacă programul/funcția ar produce același rezultat dacă ar fi apelată de mai multe ori cu aceiași parametri.

Programatorului i se oferă posibilitatea de a adnota funcții folosind cuvântul cheie "Tot", pentru a asigura că la momentul compilării se va încerca demonstrarea faptului că fiecare funcție, mai ales cele recursive, se termină, produc un rezultat și mereu același rezultat pentru aceleași argumente.

Toate metodele din modulele ce ajută la formarea rezultatului algoritmului DPLL sunt 'funcții totale', fie folosind adnotarea 'Tot' în semnatura funcției, fie prin absența oricărei adnotări, efectul 'Tot' fiind cel implicit. Succesul compilării programului înseamnă că acesta se termină și fiecare 'input' are asociat un singur rezultat. În cazul în care o formulă are mai multe soluții, modul în care a fost implementat algoritmul va decide care din acele soluții se va returna.

Acesta este un exemplu de declarație a unei funcții ce conține cuvântul cheie "Tot":

```
let rec are_variables_in_truth_assignment
  (vars: list nat_non_zero {
    L.noRepeats vars
    /\ length vars > 0 })
  (t: truth_assignment)
  : Tot
  (res : bool {
    ((all_variables_are_in_truth_assignment' vars t) <==> res) })
  (decreases length vars)
```

Se observă faptul că metoda este recursivă datorită cuvântului cheie *rec* și că este nevoie să se specifice modul în care dimensiunea parametrilor scade, de exemplu lungimea unei liste, astfel încât la un moment dat va ajunge într-un punct minim în care funcția nu se va mai apela pe sine, mai exact prin ultima linie prefixată de cuvântul cheie *decreases*.

2.3 Structurile de date folosite

S-a ales proiectarea unor structuri de date și colecții simple, pentru care au fost implementate operații generice.

Conform definiției, o formulă este o mulțime de clauze, astfel implementarea are forma unei liste înlănțuite de clauze.

```
type formula = f : list clause
```

O clauză este o mulțime de literal, de aceea structura ei apare ca o listă înlănțuită de literal.

```
type clause = c : list literal { length c > 0}
```

Un literal reprezintă o variabilă a cărei valoare de adevăr poate fi negată, motiv pentru care tipul 'literal' este creat astfel, folosind 2 constructori, 'Var' pentru a evidenția că nu se neagă valoarea de adevăr a variabilei, și 'NotVar' pentru a arăta negația variabilei.

```
type literal =
  | Var : a : nat_non_zero -> literal
  | NotVar : a : nat_non_zero -> literal
```

Doi literali, 'Var x' și 'NotVar x' sunt considerați a nu fi egali, însă au variabilele egale.

Variabilele sunt reprezentate folosind numere naturale nenule, deoarece un format des întâlnit al datelor de intrare pentru această problemă și care s-a considerat la conceperea proiectului, reprezentau literalii folosind numere întregi nenule. Am ales literalii să fie reprezentați prin doi constructori alături de numere întregi strict pozitive pentru a evita verificări repetate în cod cu privire la semnul unei variabile oarecare și pentru a se putea face ușor distincția dintre tipurile de literal.

```
let nat_non_zero = x : int {x > 0}
```

Pentru matricea de apariție a clauzelor și literalilor, s-a folosit o structură de date intermediară care reprezintă o linie a matricei și este formată dintr-o pereche de două elemente, un literal și o listă înlănțuită de clauze.

```
type aux_occurence_vector =
  {lit : literal ; clauses : list clause }

type occurence_vector = oc_v : aux_occurence_vector
  {length oc_v.clauses > 0
  /\ (forall (c : clause {L.mem c oc_v.clauses }).
    (L.mem oc_v.lit c))}
```

Condiția acestui tip de date este că lista 'clauses' trebuie să conțină toate clauzele în care apare literalul 'lit' din formula primită la începerea programului.

Matricea în sine este o listă înlănțuită de 'occurence-vector', și trebuie mereu să respecte condiția că nu există două elemente în listă, ale căror membri 'lit' să fie egali ca și valoare.

```

let rec occurrence_matrix_condition
  (oc_matrix : list occurrence_vector) =
  ( length oc_matrix > 0
    ==>
    (forall (ocv : occurrence_vector{L.mem ocv (L.tl oc_matrix)}).
      (ocv.lit <> (L.hd oc_matrix).lit))
    )
  /\ (length oc_matrix > 0 ==>
      occurrence_matrix_condition (List.Tot.tl oc_matrix))

type occurrence_matrix = oc_matrix : list occurrence_vector{
  occurrence_matrix_condition oc_matrix
}

```

Urmează un fragment în care se afișează condițiile care trebuie să fie respectate de formula 'f' și matricea de apariție a formulei notată în această funcție cu 'res'.

```

let get_occurrence_matrix
  (f : formula {length f > 0})
  : (res : occurrence_matrix{
    length res = length (get_lits_in_formula f)
    /\ (forall (l : literal{L.mem l (get_lits_in_formula f)}).
      (is_lit_in_occurrence_matrix res l))
    /\ (forall (oc_v : occurrence_vector{L.mem oc_v res}).
      (L.mem oc_v.lit (get_lits_in_formula f)) )
    /\ (forall (oc_v : occurrence_vector {L.mem oc_v res}).
      (forall (c : clause {L.mem c oc_v.clauses}).
        (L.mem c f)))
    /\ (forall (oc_v : occurrence_vector {L.mem oc_v res}).
      (forall (c : clause {
        L.mem c f /\ (L.mem c oc_v.clauses = false)}).
        (L.mem oc_v.lit c = false))))})

```

Rezultatul funcției ce verifică satisfiabilitatea unei formule este în sine o structură de date simplă creată pentru această problemă, reprezentată prin doi constructori, unul pentru situația în care formula e nesatisfiabilă și unul care conține o combinație de variabile pozitive nenule și valori booleene ce reprezintă o soluție validă pentru formulă.

```

type result =
  | NotSat
  | Sat : a: truth_assignment{ length a > 0} -> result

```

În această lucrare, se va referi la o combinație de variabile pozitive și valori booleene prin termenul 'truth-assignment' prescurtat 't' sau 'tau'.

Tipul de dată 'truth_assignment' reprezintă o listă înlănțuită de elemente, structuri formate din doi parametri, unul 'value' ce reprezintă valoarea unei variabile, și 'var' ce reprezintă în sine variabila.

```

type variable_info = {value: bool; variable: nat_non_zero}

```

```

type truth_assignment = ta : list variable_info { ta_contition ta }

```

Acest 'truth_assignment' trebuie să respecte condiția că nu există doua elemente în lista care să aibe aceeași valoare a membrului 'var'. Invariantul a fost scris astfel simplificând condiția, și anume ca membrul 'var' al elementului din capul listei trebuie să nu apară în oricare alt element din listă, iar sub-mulțimea 'tail_t', formată din toate elementele lui 't' cu excepția capului listei, trebuie să respecte aceeași proprietate.

```

let rec ta_contition (ta : list variable_info) =
  (forall (var : variable_info {List.Tot.mem var ta}).
    (count_variables_occurrence ta var.variable = 1 ))
  /\ (length ta > 0 ==> ta_contition (List.Tot.tl ta))

```

Structurile de date au fost alese a fi simple în detrimentul eficienței vitezei de execuție a programului, pentru a putea analiza dificultatea și complexitatea menținerii corectitudinii în implementarea unui algoritm complet în limbajul F*.

2.4 Modulele ce alcătuiesc programul

Separarea programului pe mai multe fișiere are scopul de a ușura dezvoltarea și modificarea codului în etapa implementării. De asemenea, orice schimbare într-un fișier rezultă în re-verificarea acestuia la momentul compilării, proces care devine cu atât mai îndelungat cu cât fișierul prezintă mai multe funcții și demonstrații complexe.

Astfel, modulele acestui program sunt:

- `DataTypes` - fișier unde s-au definit toate structurile de date explicate anterior, invariantul pentru cele ce aveau nevoie de unul, și anumite funcții ajutătoare pentru a manipula structurile de date tip colecții demonstrând în același timp corectitudinea operațiilor efectuate pe colecțiile respective.

Un exemplu de astfel de funcție ar fi metoda `'add_var_to_truth_assignment'`, al cărei simplu scop este de a adăuga un nou element în colecția `'truth_assignment'`, însa care trebuie sa poată respecta următoarea post-condiție.

```
let add_var_to_truth (t : truth_assignment)
  (var : variable_info {count_variables_occurrence t var.variable = 0})
  : Tot
  (res : truth_assignment {
    is_variable_in_assignment res var.variable
  /\ List.Tot.mem var res
  /\ count_variables_occurrence res var.variable = 1
  /\ (length res = length t + 1)
  /\ (forall (v : variable_info {
    List.Tot.mem v res /\ v <> var}). (List.Tot.mem v t))
  /\ (forall (v : variable_info {
    (List.Tot.mem v t)}). ((List.Tot.mem v res)))
  /\ (forall (v : variable_info {
    List.Tot.mem v res = false }).
    ((List.Tot.mem v t = false)))
  /\ (forall (l : literal{
    is_variable_in_assignment t (get_literal_variable l) }).
    (is_variable_in_assignment res (get_literal_variable l)
  /\ get_literal_value t l = get_literal_value res l))
```

Pe scurt, susține că noua listă trebuie să conțină toate elementele din mulțimea veche, să aibe lungimea cu 1 mai mare decât lista inițială, iar toate variabilele prezente în lista inițială au aceeași valoare în ambele mulțimi.

- Modulul `'DataTypeUtils'` conține un număr considerabil de metode ajutătoare, folosite în mai multe module ale proiectului și care trebuie deci să fie disponibile într-un singur loc.

Printre aceste metode apar:

- funcții de procesare a parametrilor primiți;

ex: *get_clause_value* - returnează valoarea de adevăr a unei clauze considerând un *'truth_assignment'* primit ca parametru;

- *lemme* care ajută la asigurarea și demonstrarea corectitudinii programului; ex: *lemma_no_vars_in_t_outside_f_length_compare* folosită pentru a arata că dacă un *'truth_assignment'* nu conține nici o variabilă care nu este prezentă în formula *'f'*, atunci sigur lungimea variabilei *'t'* este mai mică sau egală cu numărul variabilelor distincte ce apar în formula *'f'*;

```
let rec lemma_no_vars_in_t_outside_f_length_compare
  (vars: list nat_non_zero)
  (t : truth_assignment)
  : Lemma
  (requires
    L.noRepeats vars
    /\ (no_variables_outside_f_are_in_t ' vars t)
  )
  (ensures length t <= length vars )
  (decreases (length t)) =
  if length t = 0
  then let l = length vars in ()
  else let new_t =
    (remove_variable_from_assignment t (L.hd t).variable) in
  let new_vars =
    (remove_var_from_list vars (L.hd t).variable) in
  lemma_no_vars_in_t_outside_f_length_compare new_vars new_t
```

- pre/post-condiții importante și relevante demonstrării corectitudinii, referențiate folosind variabile globale pentru a reduce cantitatea de cod repetat și pentru a putea generaliza funcțiile la nivel înalt, având posibilitatea în viitor de a optimiza aspecte ale programului precum structurile de date, pentru care ar trebui modificate doar implementarea metodelor ce procesează aceste structuri;

ex: *t_cant_be_solution_for_f*, folosit ca post-condiție care enunță că orice *'truth_assignment'* ce conține toate variabilele formulei *'f'* și pentru care parametrul *'t'* este o submulțime a sa, nu este o soluție validă pentru *'f'*;

t_cant_be_solution_for_f:

```
(forall (whole_t: truth_assignment{
    t1_is_sublist_of_t2 t whole_t
    /\ are_variables_in_truth_assignment f whole_t
    /\ length whole_t
        = length ( get_vars_in_formula f ))).
(is_solution f whole_t = false))
```

- predicate, funcții care evaluează dacă anumiți parametrii respectă o anumită proprietate

ex: *is_solution*, deși simplu, este unul din cei mai importanți predicați ai programului; acesta susține că dacă un '*tau*' conține toate variabilele din formula '*f*', atunci este soluție doar dacă este soluție parțială, adică dacă nu conține clauze false.

```
let is_solution
(f: formula { length f > 0 })
(t: truth_assignment {
    ((length (get_vars_in_formula f)) = length t)
    /\ all_variables_are_in_truth_assignment f t
})
= is_partial_solution f t
```

• Modulul DpllPropagation conține puține metode însă importante pentru demonstrație, dar dificile de specificat și verificat. Aceste funcții se ocupă cu pasul de propagare a clauzelor '*unit*' din algoritmul DPLL.

O mica parte, însă cea mai importantă din specificarea metodei principale este:

```
t1_is_sublist_of_t2 t (fst res)
/\ ((t_cant_be_solution_for_f f t) <==> (
    t_cant_be_solution_for_f f (fst res)) )
/\ length res._1 >= length t
```

Post-condițiile enunță următoarele:

- ◇ faptul că tot ce e inclus in '*tau*' primit ca parametru trebuie sa fie existent și in '*tau*' trimis ca rezultat;

- ◇ lungimea 'tau'-ului rezultat trebuie sa fie măcar egală cu lungimea 'tau'-ului primit, lucru care ajută la asigurarea terminării programului, fiindcă soluția finală trebuie sa aibă atâtea elemente câte variabile unice sunt în formulă;

- ◇ 'tau'-ul primit ca parametru nu este soluție pentru formula 'f' dacă și numai dacă 'tau'-ul rezultat este soluție

- Modulul 'OccurenceMatrix' conține funcții necesare creării, procesării și accesării matricei de apariție a literalilor în clauzele formulei, și de asemenea implementarea optimizată pentru metoda ce verifică dacă un 'tau' oarecare este soluție parțială pentru formula dată.

- Modulul 'Dpll' conține funcția principală ce primește o formulă și oferă un rezultat, unde se specifică și verifică legătura dintre valoarea rezultatului și formulă.

```
: Tot (res: result {
(Sat? res = true ==>
  ((length (get-truth-from-result res) = length ( get-vars-in-formula f))
   /\ (all-variables-are-in-truth-assignment f (get-truth-from-result res))
  /\ ( is_solution f (get-truth-from-result res) = true)))
/\ ((NotSat? res = true) ==>
  ( forall (whole_t: truth_assignment{
    (forall (v : variable_info { (List.Tot.mem v pre-dpll_t)}).
      ((List.Tot.mem v whole_t)))
    /\ are-variables-in-truth-assignment f whole_t
    /\ length whole_t = length ( get-vars-in-formula f))}.
  is_solution f whole_t = false))
})
```

- Modulul 'InputFileParser' a fost conceput pentru a putea folosi formule diverse de diferite dimensiuni prin parsarea unor fișiere de intrare ce respectă un anumit format.²

- Modulul 'ConvertorToString' este folosit pentru a converti tipurile de date create pentru acest proiect în șiruri de caractere, și mai ales pentru a se afișa rezultatul pe ecran sub o forma ușoară de citit și înțeles.

²<https://people.sc.fsu.edu/~jburkardt/data/cnf/cnf.html>

2.5 Metrice orientative

Această secțiune conține metrice aproximative sub forma unor timpi ce reprezintă durata de compilare/verificare a fișierelor, sau timpul necesar obținerii rezultatului 'Satisfiabil' sau 'Nesatisfiabil'.

Algoritmul fiind unul simplu în comparație cu optimizări moderne, timpul de demonstrare că o formula este nesatisfiabilă ajunge să fie considerabil de mare chiar și pentru date de intrare cu dimensiuni relativ mici, motiv pentru care sunt necesare îmbunătățiri ale algoritmului pentru a ajunge la o eficiență comparabilă.

Compilerul a avut nevoie de aproximativ 500 de secunde pentru verificarea fișierelor '.fst' și extragerea fișierelor '.ml' specifice limbajului Ocaml.

Nume fișier	Durata verificare/compilare
DataTypes	8 secunde
DataTypeUtils	33 secunde
DpllPropagation	285 secunde
OccurenceMatrixUtils	91 secunde
Dpll	71 secunde
ConvertorToString	3 secunde
InputFileParser	4 secunde
Main	3 secunde

Tabelul următor ilustrează timpii necesari obținerii unor rezultate.

Nume fișier input	Tip rezultat	Durata obținerii rezultatului
hole6.cnf	UNSAT	23 secunde
aim-50.cnf	SAT	< 0.1 secunde
aim-100.cnf	UNSAT	câteva minute
quinn.cnf	SAT	< 0.1 secunde

Analizând codul sursă, se observă că fișierele ce conțin mai multe linii de cod pentru procesarea datelor și mai puține ce necesită demonstrarea unor proprietăți simple, au nevoie de mai puțin timp pentru a se verifica și compila. În schimb, fișierele care au foarte puține linii de cod ce contribuie la formarea rezultatului și care ajung să fie extrase în fișierele '.ml', dar multe proprietăți mai complexe ale căror demonstrație trebuie verificată, necesită mult mai mult timp de compilare.

Dispozitivul pe care s-a făcut această contorizare avea următoarele specificații:

- cpu - intel-core i5-1035G1
- memorie ram - 16gb
- tip hard-disk stocare - ssd

2.6 Posibile optimizări

Mulți 'SAT solveri' moderni implementează algoritmul CDCL (Conflict-Driven-Clause-Learning) ³ care, pe lângă pașii din algoritmul DPLL, adaugă un pas nou de învățare și eficientizează saltul înapoi în momentul în care se găsește o soluție falsă.

Un alt rafinament pe lângă îmbunătățirea algoritmului implementat poate fi eficientizarea structurilor de date folosite, pentru a reduce complexitatea timp a operațiilor ce se repetă cel mai des.

O problemă ce poate apărea și în implementarea algoritmului DPLL dar și cea a algoritmului mai complex CDCL, este o metoda/euristica ineficientă de a alege o nouă variabilă pentru a i se asigura o valoare, și însăși valoarea care să i se asocieze. Un exemplu ar fi metoda VSIDS ⁴, care se bazează pe asocierea unui scor pentru fiecare variabilă, scor care se schimbă dacă variabila apare într-un conflict.

O altă optimizare ce se poate implementa este paralelizarea calculelor și împărțirea ramurilor/soluțiilor ce trebuie explorate.

O mică îmbunătățire, din perspectiva limbajului de programare de aceasta dată, poate fi detectarea instrucțiunilor ce au rolul de a ajuta demonstrarea unor proprietăți însă ajung pe parcursul dezvoltării programului să devină redundante, iar eliminarea lor ar putea reduce puțin timpii de validare/compilare.

De asemenea, mai există varianta de a simplifica problema pe care algoritmul trebuie să o rezolve, iar analiza fiecărei clauze în parte, poate duce la identificarea unor clauze și literalii redundanți, care pot fi eliminați din formula primită fără a schimba completitudinea și corectitudinea rezultatului final.

³<https://www.cs.princeton.edu/~zkincaid/courses/fall18/readings/SATHandbook-CDCL.pdf>

⁴https://mk.cs.msu.ru/images/1/1f/SAT_SMT_Vijay_Ganesh_HVC2015.pdf

Capitolul 3

Pași necesari pentru a reproduce

Informații despre replicarea condițiilor necesare execuției programului scris folosind F* se pot găsi pe pagina de github a limbajului F*. ¹

În secțiunea următoare se prezintă pașii care au fost luați pentru crearea mediului în care s-a dezvoltat 'SAT solver-ul'. Instrucțiunile următoare sunt compatibile cu sistemul de operare Windows, verificat cu versiunile 10/11.

3.1 Programe si resurse necesare

Următoarele aplicații/programe/resurse trebuie descărcate de la locația specificată fiecăreia în fișierul 'INSTALL.md' găsit pe pagina de github a limbajului FStar.

- OCaml - necesar compilării și executării fișierelor OCaml (.ml), care rezultă în urma compilării fișierelor FStar (.fst) (versiune folosită - 4.14.0)
- opam - folosit pentru a instala pachetele necesare compilării fișierelor specifice limbajului de programare OCaml
(versiunea folosită pentru lucrare - 2.0.10)
- cygwin - oferă posibilitatea compilării și executării a programelor tipice sistemelor de operare Unix și Linux, ceea ce include suport pentru fișiere 'Makefile' (versiunea folosită pentru lucrare - 3.4.6)
- Z3 - folosit pentru a valida fișierele ce conțin programe scrise folosind F*
(arhiva folosită pentru Windows - z3-4.8.5-x64-win.zip)

¹<https://github.com/FStarLang/FStar/blob/master/INSTALL.md>

După descărcarea/instalarea acestor resurse, trebuie clonată ramura 'master' a proiectului FStar pe dispozitivul local. (locația clonei pe dispozitivul folosit pentru acest proiect: "D:/fstar", versiunea - F* 2023.04.26 dev)

Trebuie adăugate path-urile absolute către ".../fstar/bin" și ".../z3-win/bin" în variabila 'Path' a sistemului.

După instalarea programului 'opam', trebuie instalate anumite pachete de date. Minimul necesar de pachete se poate găsi și pe instrucțiunile de instalare găsite la link-ul de mai sus, însă pentru a avea la dispoziție toate resursele din proiectul FStar descărcat fără erori, sunt necesare următoarele pachete:

# Name	# Installed	# Synopsis
base-bigarray	base	
base-bytes	base	Bytes library distributed with the OCaml compiler
base-threads	base	
base-unix	base	
batteries	3.5.1	A community-maintained standard library extension
conf-gmp	4	Virtual package relying on a GMP lib system installation
cppo	1.6.9	Code preprocessor like cpp for OCaml
csexp	1.5.1	Parsing and printing of S-expressions in Canonical form
depxt	transition	opam-depxt transition package
depxt-cygwinports	0.0.9	obsolete depxt wrapper for windows
dune	3.5.0	Fast, portable, and opinionated build system
dune-configurator	3.5.0	Helper library for gathering system configuration
gen	1.0	Iterators for OCaml, both restartable and consumable
menhir	20220210	An LR(1) parser generator
menhirLib	20220210	Runtime support library for parsers generated by Menhir
menhirSdk	20220210	Compile-time library for auxiliary tools related to Menhir
num	1.4	The legacy Num library for arbitrary-precision integer and rational arithmetic
ocaml	4.14.0	The OCaml compiler (virtual package)
ocaml-compiler-libs	v0.12.4	OCaml compiler libraries repackaged
ocaml-config	2	OCaml Switch Configuration
ocaml-variants	4.14.0+mingw64c	Pre-compiled 4.14.0 release (mingw64)
ocamlbuild	0.14.2	OCamlbuild is a build system with builtin rules to easily build most OCaml projects
ocamlfind	1.9.5	A library manager for OCaml
opam-depxt	1.1.5	Install OS distribution packages
pprint	20220103	A pretty-printing combinator library and rendering engine
ppx_derivers	1.2.1	Shared [@@deriving] plugin registry
ppx_deriving	5.2.1	Type-driven code generation for OCaml
ppx_deriving_yoison	3.7.0	JSON codec generator for OCaml
ppxlib	0.28.0	Standard library for ppx rewriters
process	0.2.1	Easy process control
result	1.5	Compatibility Result module
sedlex	3.0	An OCaml lexer generator for Unicode
seq	base	Compatibility package for OCaml's standard iterator type starting from 4.07.
sexplib0	v0.15.1	Library containing the definition of S-expressions and some base converters
stdint	0.7.2	Signed and unsigned integer types having specified widths
stdlib-shims	0.3.0	Backport some of the new stdlib features to older compiler
uchar	0.0.2	Compatibility library for OCaml's Uchar module
yoison	2.0.2	Yoison is an optimized parsing and printing library for the JSON format
zarith	1.12	Implements arithmetic and logical operations over arbitrary-precision integers

La finalul acestor pași, folosind terminalul Cygwin și instrucțiunile de tipul 'make' în folder-ul 'fstar', ar trebui să funcționeze verificarea și executarea oricăror fișiere surse scrise în F*, fișiere proprii sau exemple ce făceau deja parte din proiect.

3.2 Executarea solver-ului SAT

Sursele corespunzătoare proiectului prezentat se găsesc la: FStar-DPLL github.

Aceste surse trebuie descărcate, salvate într-un folder în proiectul 'fstar'.

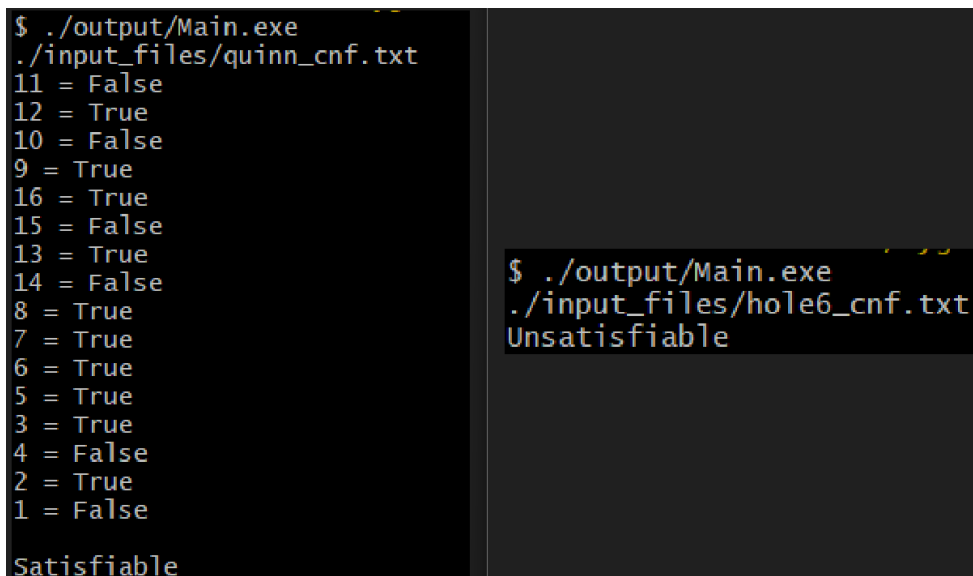
Fișierul 'Makefile' trebuie modificat, astfel încât variabila 'FSTAR-HOME' să facă referire spre folder-ul 'fstar'. Același pas trebuie făcut pentru fișierul 'Makefile' din folder-ul 'output'.

Apoi, în terminalul cygwin deschis în folder-ul proiectului DPLL-FStar, trebuie executată comanda 'make', la finalul căreia în folder-ul 'output' vor apărea pentru fiecare fișier sursă '.fst', câte un fișier '.ml', iar acestea conțin codul Ocaml extras din sursele FStar. De asemenea în folder-ul 'output' se va afla executabilul "Main.exe".

Pentru a recompila și regenera fișierul "Main.exe", trebuie șters cel anterior creat, dacă a fost creat.

Imediat după pornirea programului "Main.exe", trebuie introdus de la tastatură calea relativă către un fișier de input. Câteva fișiere de input există în folder-ul "input-files" și orice alt fișier de intrare trebuie să respecte acea structură pentru ca parsarea implementată a datelor să funcționeze.

La finalul unei astfel de execuții, va apărea mesaj la consolă cu rezultatul obținut, fie că formula dată este nesatisfiabilă, fie că e satisfiabilă și alături o variantă de răspuns ce conține variabilele formulei și valorile lor booleene astfel încât fiecare clauză a formulei să aibe valoarea de adevăr *true*.



```
$ ./output/Main.exe
./input_files/quinn_cnf.txt
11 = False
12 = True
10 = False
9 = True
16 = True
15 = False
13 = True
14 = False
8 = True
7 = True
6 = True
5 = True
3 = True
4 = False
2 = True
1 = False
Satisfiable

$ ./output/Main.exe
./input_files/hole6_cnf.txt
Unsatisfiable
```

Concluzii

Prin urmare, deși acest 'solver' nu aplică multe tehnici de optimizare sau cele mai eficiente structuri de date, prezintă cum se poate folosi sistemul de verificare formală și demonstrare a limbajului FStar pentru algoritmul DPLL. Astfel toate specificațiile ce corespund verificării algoritmului DPLL pot funcționa ca o bază pentru specificațiile oricărei extensii ale metodei DPLL, precum CDCL sau alte variante optimizate.

Solver-ul este dovedit a fi complet, corect, garantat că produce un rezultat, iar generalizarea modulelor și interacțiunii dintre ele poate însemna că este un bun punct de plecare pentru crearea unui solver mai eficient folosind limbajul FStar, aplicând îmbunătățiri precum cele prezentate pe scurt anterior în această lucrare.

Bibliografie

- despre problema SAT:
https://en.wikipedia.org/wiki/Cook%E2%80%93Levin_theorem
- tutorial fstar:
<https://www.fstar-lang.org/tutorial/>
- github fstar:
<https://github.com/FStarLang/FStar>
- despre 'sat solveri':
<https://www.borealisai.com/research-blogs/tutorial-9-sat-solvers-i-introduction-and-applications/>
- istoric al problemei SAT, descrierea problemei, algoritmului DPLL, si unor extensii ale sale :
Book: Handbook of Satisfiability Armin Biere, Marijn Heule, Hans van Maaren and Toby Walsh (Eds.)
<http://gauss.eecs.uc.edu/Courses/c626/notes/history.pdf>
- fisiere de intrare: <https://people.sc.fsu.edu/~jburkardt/data/cnf/cnf.html>
- sursele proiectului:
https://github.com/alex4482/FStar-DPLL-licenta/tree/main/dpll_optimized