Lista zadań pracownianych nr 5

Spełnialność formuł logicznych w CNFie: rezolucja

W ćwiczeniu 6 zobaczyliśmy że nie jest trudno sprawdzić czy formuła w koniunkcyjnej postaci normalnej jest tautologią (i znaleźć kontrprzykład jeśli nią nie jest). Znacznie trudniejszym problemem, o bardzo istotnym znaczeniu praktycznym, jest problem *spełnialności* takich formuł, czyli *istnienia* wartościowania spełniającego daną formułę.

Problem spełnialności formuł rachunku zdań jest o tyle ciekawy, że wiele praktycznych problemów występujących w informatyce (od reprezentacji i poprawności układów logicznych po zagadnienia związane z automatyczną weryfikacją programów) można przedstawić jako instancje tego problemu. Jest on jednak trudny obliczeniowo: jest sztandarowym przedstawicielem rodziny-problemów dla których potrafimy efektywnie *sprawdzić* rozwiązanie (mając dane wartościowanie umiemy sprawdzić czy spełnia ono formułę), ale nie umiemy go efektywnie *znaleźć* (co więcej, większość teoretyków uważa że nie da się tego zrobić efektywnie).[1]

Nie znaczy to jednak że nie możemy próbować! Wymyślono wiele technik pozwalających na relatywnie efektywne poszukiwanie wartościowania spełniającego daną formułę (lub stwierdzenie że jest ona sprzeczna). Dziś zajmiemy się implementacją jednej z ciekawszych z nich: rezolucji.

Rezolucja w rachunku zdań

Rezolucję dla rachunku zdań możecie pamiętać z poprzedniego semestru, z logiki dla informatyków. Dla tych którzy nie pamiętają, poniżej zamieszczamy krótkie przypomnienie.

Aby zdefiniować potrzebne pojęcia, przyjmijmy że przedstawiamy formułę logiczną w CNFie jako zbiór klauzul, z których każda jest zbiorem literałów (kon-

[1] Oczywiście, trywialnie można sprawdzić spełnialność formuły w DNFie: problemem jestsprowadzenie formuły do DNFu. Translacje do równoważnych formuł w CNFie lub DNFie mogą wykładniczo zwiększyć rozmiar formuły, ale do CNFu łatwo przetłumaczyć formułę tak żeby zachować jej spełnialność (używając świeżych zmiennych) — podczas gdy dla DNFu jest to równie trudne jak problem spełnialności CNFu.

kretną reprezentacją w Rackecie zajmiemy się później, tę przyjmujemy po stronie matematycznej). Kluczową obserwacją jest że zbiór klauzul $\{c_1 \vee p, c_2 \vee \neg p\}$ (gdzie p nie występuje w c_1 ani c_2) jest spełnialny wtedy i tylko wtedy gdy spełnialna jest klauzula $c_1 \vee c_2$. Klauzulę $c_1 \vee c_2$ nazywamy w takim wypadku *rezolwentą* klauzul $c_1 \vee p$ i $c_2 \vee \neg p$ względem zmiennej p.

Używając operacji rezolucji możemy budować dowody wyprowadzalności przez rezolucję pewnej klauzuli z danego zbioru klauzul. Na przykład, mając dany zbiór klauzul $\{\neg p \vee q, p \vee q, \neg q \vee r, \neg q \vee \neg r\}$, możemy wyprowadzić z niego przez rezolucję klauzulę p w następujący sposób:

$$\dfrac{\dfrac{p \vee q \qquad \dfrac{\neg q \vee r \qquad \neg q \vee \neg r}{\neg q}\, r}{p}\, q}{}$$

gdzie pozioma linia oznacza wyprowadzenie z klauzul bezpośrednio nad nią klauzuli bezpośrednio pod nią przez rezolucję względem zmiennej obok linii, zaś liśćmi są klauzule należące do rozważanego zbioru.

Szczególnym rodzajem dowodu wyprowadzalności jest rezolucyjny dowód sprzeczności, czyli dowód który kończy się klauzulą pustą (oznaczaną przez $\bot$). W przypadku naszego zbioru, taki dowód można uzyskać jako:

$$\dfrac{\dfrac{p \vee q \qquad \neg p \vee q}{q}\, p \qquad \dfrac{\neg q \vee r \qquad \neg q \vee \neg r}{\neg q}\, r}{\bot}\, q$$

Łatwo pokazać że jeśli rezolucyjny dowód sprzeczności istnieje, to początkowy zbiór klauzul jest sprzeczny (robiliście to na ćwiczeniach z logiki!) — ale żeby przekonać się o tym w przypadku konkretnego dowodu trzeba umieć sprawdzić że jest on poprawny.

Ćwiczenie 1. Poprawność dowodu rezolucyjnego

Pierwszym zadaniem jest napisanie procedury sprawdzającej poprawność rezolucyjnego dowodu sprzeczności. W tym celu musimy umieć takie dowody jakoś reprezentować. Szczęśliwie, jak widzimy, są one drzewami pewnego specjalnego rodzaju, więc nie powinny sprawić nam szczególnego problemu.

Zacznijmy od reprezentacji formuł w CNFie. Zmienne reprezentujemy jako symbole, podobnie jak w ogólnych formułach logicznych. Literały reprezentujemy jako listy długości 3 tagowane symbolem `'lit`, których drugim elementem jest prawda gdy zmienna jest niezanegowana, lub fałsz gdy literał oznacza

zmienną zanegowaną. Przykładowo, listy `'(lit #t p)` i `'(lit #f q)` oznaczają odpowiednio literały p i $\neg q$. Klauzule reprezentujemy jako listy literałów poprzedzone tagiem `'clause`, zaś formuły jako listy klauzul poprzedzone tagiem `'cnf`. Zatem zbiór klauzul rozważany powyżej może mieć następującą postać:

```
'(cnf (clause (lit #f p) (lit #t q))
      (clause (lit #t p) (lit #t q))
      (clause (lit #f q) (lit #t r))
      (clause (lit #f q) (lit #f r)))
```

Drzewem wyprowadzenia będzie albo liść, składający się z klauzuli opatrzonej tagiem `'axiom`, albo wierzchołek wewnętrzny, reprezentowany jako lista czteroelementowa składająca się z tagu `'resolve`, zmiennej względem której wykonujemy rezolucję, drzewa wyprowadzenia klauzuli w której nasza zmienna występuje pozytywnie i drzewa wyprowadzenia klauzuli w której zmienna występuje negatywnie. Odpowiednie predykaty to:

```
(define (axiom? p)
  (tagged-tuple? 'axiom 2 p))

(define (res? p)
  (tagged-tuple? 'resolve 4 p))

(define (proof? p)
  (or (and (axiom? p)
           (clause? (axiom-clause p)))
      (and (res? p)
           (var? (res-var p))
           (proof? (res-proof-pos p))
           (proof? (res-proof-neg p)))))
```

W szablonie zadania znajdują się oczywiście również odpowiednie konstruktory i selektory. Poprawną reprezentacją dowodu sprzeczności pokazanego wyżej będzie zatem:

```
'(res q
      (res p (axiom (clause (lit #t p) (lit #t q)))
             (axiom (clause (lit #f p) (lit #t q))))
      (res r (axiom (clause (lit #f q) (lit #t r)))
             (axiom (clause (lit #f q) (lit #f r)))))
```

Waszym zadaniem jest zdefiniowanie procedury `proof-result` przyjmującej dowód i formułę w CNFie, i zwracającej klauzulę powstającą w korzeniu drzewa lub `#f` jeśli dowód nie jest poprawny. Wówczas sprawdzenie poprawności rezolucyjnego dowodu sprzeczności sprowadza się do uruchomienia tej procedury i sprawdzenia czy jej wynik jest klauzulą pustą:

```
(define (check-proof? pf prop)
  (let ((c (proof-result pf prop)))
    (and (clause? c)
         (null? (clause-lits c)))))
```

Zdefiniowaną procedurę należy oczywiście przetestować definiując odpowiedni zestaw testów o nazwie `proof-checking-tests`.

Znajdowanie dowodu rezolucyjnego

Wiemy już że używając rezolucji możemy udowodnić sprzeczność danego zbioru klauzul, pozostaje jednak pytanie jak taki dowód znaleźć i jak stwierdzić że dowód nie istnieje gdy formuła jest spełnialna. W tym celu zauważmy, że jeśli mamy zbiór klauzul X i klauzule $c_1, c_2 \in X$ których rezolwentą jest c, to X jest spełnialne wtedy i tylko wtedy gdy $X \cup \{c\}$ jest spełnialne. Ta obserwacja pozwala nam zdefiniować (na razie nieformalnie) operację $F(X)$, która ze zbioru klauzul buduje potencjalnie większy zbiór, zawierający klauzule wyprowadzalne z X przez rezolucję:

$$F(X) = X \cup \{c \mid \exists c_1, c_2 \in X.\ \text{resolve}(c_1, c_2) = c\}$$

Możemy teraz znaleźć najmniejszy *punkt stały* operatora F zawierający X — podobnie jak w przypadku pierwiastkowania metodą Newtona! — czyli taki zbiór Y, że $X \subseteq Y$ i $F(Y) = Y$, czyli że przez rezolucję nie da się do zbioru Y dodać żadnych nowych klauzul. Łatwo zauważyć, że znajdując Y możemy odpowiedzieć na nasze pytanie: zbiór X jest spełnialny wtedy i tylko wtedy gdy zbiór Y jest spełnialny, a ten jest spełnialny wtedy i tylko wtedy gdy nie zawiera klauzuli pustej $\bot$![2] Wystarczy teraz intuicje zawarte powyżej ubrać w reprezentację programistyczną aby dostać działającą implementację znajdowania rezolucyjnych dowodów sprzeczności.

Zakładając że mamy odpowiednik powyższego operatora F, jednoargumentową procedurę `improve` zwracającą reprezentację rozszerzonego zbioru lub `#f` gdy osiągnięto punkt stały, możemy zbudować rozwiązanie w następujący sposób:

```
(define (fixed-point op start)
  (let ((new (op start)))
    (if (eq? new false)
        start
        (fixed-point op new))))
```

[2]Dowód jednego z kierunków jest prosty, drugiego trochę trudniejszy, oba w tym miejscu pomijamy.

```
(define (prove cnf-form)
  (let* ((clauses (cnf->clause-set cnf-form))
         (sat-clauses (fixed-point improve clauses))
         (pf-or-false (get-empty-proof sat-clauses)))
    (if (eq? pf-or-false false)
        'sat
        (list 'unsat pf-or-false))))
```

Powyższa definicja zwraca albo symbol `'sat` (gdy nie znaleziono klauzuli pustej), albo dowód wyprowadzenia tej klauzuli z oryginalnego zbioru klauzul uzyskany dzięki procedurze `get-empty-proof`. Procedura ta, podobnie jak `improve` i `cnf->clause-set` używa wewnętrznej reprezentacji zbiorów klauzul, która musi pozwalać na odtworzenie dowodów wyprowadzenia i wygodną implementację rezolucji, i którą musimy dopiero zdefiniować.

Wewnętrzna reprezentacja klauzul i zbiorów klauzul

Wejściowa reprezentacja klauzul, którą można wygodnie wprowadzać i ładnie wypisywać, nie jest niestety najlepsza do szukania dowodów rezolucyjnych. Wygodniej reprezentować klauzulę przez dwie listy, zawierające odpowiednio zmienne występujące w danej klauzuli pozytywnie i negatywnie. Dodatkowo, nowa reprezentacja jest przydatna, gdyż możemy przechować w niej rezolucyjny dowód wyprowadzenia naszej klauzuli z bazowego zbioru. Reprezentacja jest dana następującym predykatem (`sorted-varlist?` jest predykatem mówiącym że jego argument jest posortowaną alfabetycznie listą zmiennych):

```
(define (res-clause? p)
  (and (tagged-tuple? 'res-clause 4 p)
       (sorted-varlist? (second p))
       (sorted-varlist? (third  p))
       (proof? (fourth p))))
```

Przykładowo, reprezentacją klauzuli $p \vee q \vee \neg r$, wyprowadzonej przez dowód `pf` jest `(list '(p q) '(r) pf)`.

Ostatnim problemem na który nie zwracaliśmy do tej pory uwagi jest jak wybrać parę klauzul których ewentualną rezolwentę chcielibyśmy dorzucić do naszego zbioru. W tym celu przyjmiemy następujący interfejs zbiorów klauzul, który dzieli klauzule na te już przetworzone i te które jeszcze musimy przetworzyć:

- wartość `clause-set-empty` oznacza pusty zbiór klauzul
- predykat `clause-set-done?` sprawdza czy cały zbiór został przetworzony;
- procedura `clause-set-next-pair` zwraca trójkę zawierającą nierozważaną wcześniej parę klauzul i zbiór w którym zwrócona para klauzul jest oznaczona jako przetworzona;

- procedura `clause-set-add` dodaje nową, nieprzetworzoną klauzulę do zbioru, sprawdzając czy klauzula ta już w nim nie występuje;
- procedura `clause-set-done->clause-list` pozwala odzyskać z przetworzonego w całości zbioru wszystkie klauzule

Teraz łatwo zdefiniować procedury z poprzedniej sekcji:

```
(define (cnf->clause-set f)
  (define (aux cl rc-set)
    (clause-set-add (clause->res-clause cl) rc-set))
  (foldl aux clause-set-empty (cnf-clauses f)))

(define (get-empty-proof rc-set)
  (define (rc-empty? c)
    (and (null? (rc-pos c))
         (null? (rc-neg c))))
  (let* ((rcs (clause-set-done->clause-list rc-set))
         (empty-or-false (findf rc-empty? rcs)))
    (and empty-or-false
         (rc-proof empty-or-false))))

(define (improve rc-set)
  (if (clause-set-done? rc-set)
      false
      (let* ((triple (clause-set-next-pair rc-set))
             (c1     (car  triple))
             (c2     (cadr triple))
             (rc-set (cddr triple))
             (c-or-f (rc-resolve c1 c2)))
        (if (and c-or-f (not (rc-trivial? c-or-f)))
            (clause-set-add c-or-f rc-set)
            rc-set))))
```

Ćwiczenie 2.

W powyższej implementacji brakuje definicji procedur `rc-resolve`, zwracającej rezolwentę dwóch klauzul (lub `#f` jeśli ta nie istnieje) i `rc-trivial?` sprawdzającej czy klauzula jest trywialna (tj. spełnialna dla każdego wartościowania — takiej klauzuli oczywiście nie ma sensu rozpatrywać w naszym zbiorze). Rozwiązanie przetestuj: styl rozwiązania i jakość testów ma w tym zadaniu duże znaczenie, rozwiązania słabo przetestowane lub nieczytelne nie będą punktowane!

Rozwiązania obydwu zadań należy przesłać w systemie SKOS w jednym pliku, zgodnym z dołączonym szablonem, do poniedziałku, 1 kwietnia 2019 r., godz. 06.00.