LIETUVOS MATEMATIKOS RINKINYS
Proc. of the Lithuanian Mathematical Society, Ser. A
Vol. 62, 2021, pages 16–22
https://doi.org/10.15388/LMR.2021.25215

Logical derivation search with assumption traceability

Adomas Birštunas, Elena Reivytytė

Faculty of Mathematics and Informatics, Vilnius University
Naugarduko 24, LT-03225 Vilnius
E-mail: adomas.birstunas@mif.vu.lt; e.reivytyte@gmail.com

Received September 1, 2021; published online December 15, 2021

Abstract. In this paper authors research the problem of traceability of assumptions in logical derivation. The essence of this task is to trace which assumptions from the available knowledge base of assumptions are necessary to derive a certain conclusion. The paper presents a new derivation procedure for propositional logic, which ensures traceability feature. For the derivable conclusion formula derivation procedure also returns the smallest set of assumptions those are enough to get derivation of the conclusion formula. Verification of the procedure were performed using authors implementation.

Keywords: propositional logic; traceability; loop checking

Introduction

In the recent decades artificial intelligence is increasingly used by software developers to provide more and more smart systems. One of the areas of the artificial intelligence is formal logics. There are lots of classical methods for logical derivation. Some of them are applicable for particular logic, some of them are universal methods, those may be applied for a big set of logics. The most widely used methods are sequent calculi, resolution methods, tableaux methods [5]. The logic to be chosen depends on the application scope and may vary from the propositional logic, predicate logic, descriptive logic and different modal logics. The goal of these methods is to determine if given formula (conclusion) is derivable from the set of assumption formulas. Different provers based on the different calculi or derivation procedures are already implemented and used in real word applications.

In the real applications we use knowledge bases, those contain hundreds, thousands or even much more assumptions. Most of the assumptions are not related to the particular goal formula. Usually prover, especially if we take a general-purpose prover, returns simple result – if given conclusion formula is derivable or non-derivable. For the most applications such a simple result is enough. For some application we also need the smallest set of assumptions, those are enough to get a derivation of given conclusion formula.

For example, suppose we have logic-based intelligent system for reasoning on requirements of legal acts. Our knowledge base is a set of assumptions (logic formulas), those describe particular requirements, restrictions presented in the legal acts. Suppose, our system may check if some given restriction, requirement shall be applied or not, or some action, operation is legal or not in the particular situation. Using classical prover, we will be able to provide the main answer – is it applicable or not. But in the example case, it is not enough. The system user should be able to explain the decision correctness – to present the set of legal act requirements, those leads to the decision. Therefore, we also need the set of assumptions (related to particular legal requirements) used to derive our conclusion.

General purpose calculi and decision procedures do not care about assumption traceability. Usually they may by modified to return used assumptions. Unfortunately, the returned set of assumptions may contain unnecessary assumptions, those are not needed for the derivation, but they were used during derivation. There can be few different derivation trees based on the different assumptions, classical methods will return only one of them. As a result, we may get the set of used assumptions containing 50 or 100 assumptions, even if there exist a small set of 5 or 10 assumptions, those are enough to prove a goal formula. Therefore, we need methods those ensure assumption traceability feature – returning the smallest set of assumptions for provided formula derivation.

There is also other research area of the traceability. When a set of used assumptions to derive some goal and derivation tree is already presented, we may analyse the decision tree itself to get even more useful information. Sometimes formal logics are also used for such a reasoning. Such a research may be found in [3].

In the next section we will introduce a new propositional logic method with assumption traceability feature. We will call it “Decision procedure with assumption traceability for the propositional logic”, or shortly PWATPL procedure. For creating PWATPL procedure some techniques were used – indexing formulas and loop checking. Indexing formulas are widely used in logical calculi for different modal logics. Usually indexing formulas are used to show finality, completeness or soundness of the calculus (see [4, 6]). We use indexed formulas for the different purpose. We will introduce 2 level indexes: the first is used to ensure the smallest set of assumptions, the second – to ensure traceability itself. Loop checking technique for some modal logics are used to achieve the finality of the calculus (see [1]). We will use loop checking to ensure completeness and soundness of our method.

1 Derivation procedure

Suppose we have the set of premises $\{\varphi_1, \dots, \varphi_n\}$ (assumption formulas) and the goal formula ψ.

Definition 1. Suite is a structure: $[i_1, i_2, i_3, \dots, i_n; \mathit{state}; q_1, q_2, \dots, q_k]$, where:

- $i_1, i_2, i_3, \dots, i_n$ – 2-level clause indices.
- $\mathit{state} \in \{\mathit{closed}, \mathit{loop}\}$ – the state of the suite.
- $q_1, q_2, \dots, q_k$ – looping literals of this suite; not used for closed suites.

Example 1. Suite $[2.2, 3.1, 3.2; \mathit{loop}; p, \neg q]$ is a loop suite (with looping literals p and $\neg q$), containing indeces, those refer to clauses $D_{2.2}, D_{3.1}, D_{3.2}$ from 2 premises φ_2, φ_3.

Definition 2. Joint (union) suite of the suites $S_1 = [i_1, i_2, \dots, i_n; st_1; q_1, q_2, \dots, q_k]$ and $S_2 = [j_1, j_2, \dots, j_m; st_2; p_1, p_2, \dots, p_r]$ is a new suite $S = S_1 \cup S_2 = [\{i_1, i_2, \dots, i_n\} \cup \{j_1, j_2, \dots, j_m\}, st, \{q_1, q_2, \dots, q_k\} \cup \{p_1, p_2, \dots, p_r\}]$, where state $st = \mathit{closed}$ if $st_1 = st_2 = \mathit{closed}$, and, otherwise, $st = \mathit{loop}$.

Definition 3. Literal structure $LS = \langle \mathit{state}, \mathit{setOfSuites} \rangle$, where:

- $\mathit{state} \in \{\mathit{init}, \mathit{started}, \mathit{finished}\}$ – current state.
- $\mathit{setOfSuites}$ – set of suites assigned to the literal.

Initialization:

1) Convert every premise into conjunctive normal form (CNF): $\varphi_i = D_{i.1} \& D_{i.2} \& \dots \& D_{i.m}$, where $D_{i.j}$ is a disjunctive clause (further *clause*).
2) For every obtained clause $D_{i.j}$ give a two level index – $i.j$. The first index part refers to the premise, and the second part refers to the particular clause.
3) Create set S of all clauses (clause indexes) obtained from every premise.
4) For every literal used in any clause create a structure LS.

PWATPL procedure for the goal formula G derivation follows:

1) Convert goal formula G into disjunctive normal form (DNF): $G = F_1 \vee F_2 \vee \dots \vee F_s$, where F_l conjunctive clause (further *conjunct*).
2) For every conjunct F_l perform derivation procedure $\mathit{deriveConjunct}(F_l)$.
3) Put obtained suite into the set of good suites – $\mathit{GoodSuites}$.
4) Choose the smallest suite from the $\mathit{GoodSuites}$. Note: only the first level indices should be taken into consideration during suite comparison, since they define real premises, not their parts only.

Conjunct $F = p_1 \& p_2 \& \dots \& p_k$ derivation procedure $\mathit{deriveConjunct}(F)$ follows:

1) For every literal p_j perform procedure $\mathit{getSuitesFor}(p_j)$. As the result we get the set of suites $R'(p_j)$ for the literal p_j. $R'(p_j)$ will contain closed suites and loop suites as well.
2) Obtain the set of the closed suites $R(p_j) \subset R'(p_j)$ – remove every loop suite from $R'(p_j)$ and leave only the closed ones. $R(p_j)$ contains suites, from which literal p_j is derivable.
3) Join all the obtained set of suites for single literals p_j $(j = 1, 2, \dots, k)$ into the one set of closed suites $R(F)$ for conjunct $F = p_1 \& p_2 \& \dots \& p_k$.

$R(F)$ is obtained similar to getting Cortesian product – every suite form $R(p_1)$ should be joint (union operation) with every suite from $R(p_2)$, and the result suites – with every suite from $R(p_2)$, and so on. If $R(p_j) = \emptyset$, for any j, $R(F)$ is also an empty set. $R(F)$ contains suites, from which conjunct F is derivable.

4) Choose the smallest suite from the set $R(F)$. Note: only the first level indices should be taken into consideration during suite comparison, since they define real premises, not their parts only.
5) The smallest suite from the set $R(F)$ is a result of *deriveConjunct(F)*.

Now we introduce helper suite generation procedure $genSuites(suiteMap, C(p))$. It generates suites for the literal p from suites of the literals containing in the same clause.

Suppose clause $C(p) = p \vee \neg c_1 \vee \ldots \neg c_m$, and suite map for literals is $suiteMap = \{c_1 \to \{S_{1.1}, \ldots, S_{1.k_1}\}, \ldots, c_m \to \{S_{m.1}, \ldots, S_{m.k_m}\}\}$. $\{S_{x.1}, \ldots, S_{x.k_x}\}$ is a set of the suites of the literal c_x.

Procedure $genSuites(suiteMap, C(p))$ generates new suites for literal p as follows:

1) Create set of union suites
$S = \{S_{1.1} \cup S_{2.1} \cup \cdots \cup S_{m.1}, \ldots, S_{1.k_1} \cup S_{2.k_2} \cup \cdots \cup S_{m.k_m}, \}$ – all posible unions of m suites taking one suite per one literal (like Cartesian product).
2) Add $C(p)$ index into every suite of the set S.
3) Remove litaral p from looping literals from every suite of the set S.
4) From the suite set S remove duplicates and supersuites (leave only subsuites).

Finally, we present the main procedure $getSuitesFor(p_j)$. It returns the set of suites literal p_j is derivable from.

For the further text, by the $\neg c$ we denote the opposite literal to c: if $c = q$, then $\neg c = \neg q$, if $c = \neg q$, then $\neg c = q$. Therefore, $\neg c$ does not mean, that literal has negation.

Procedure $getSuitesFor(p_j)$ follows:

```
path.add($p$);                            | 1. add literal p into the derivation path
if ( $p.explored$ )                       | 2. if literal p is already explored
  return $p.suites$;                      | just return previously found suites
$p.state$ = started;                      | 3. initialize process for literal p
for ($C(p)$: getClauseWith($p$))          | 4. for every clause containing p proceed:
begin                                     |
  if (path.contains($C(p)$))              | 4.1. if clause is on the derivation path
    continue;                             | continue with a next clause for literal p,
                                          | go to step 4
  if ( $C(p) = p$ )                       | 4.2. if clause C(p) is p – it is derivable:
  begin                                   |
    $suite$ = new [$C(p)$.index; closed;];| – create new closed suite
                                          | containing only C(p) index
    $p$.suites.add( $suite$ );            | – add new suite for the literal p
    continue;                             | – continue with the next clause
  end                                     | and go to step 4
  path.add( $C(p)$ );                            | 4.3. add clause $C(p)$ to the path
  $suiteMap$ = new Map(literal to set);          | 4.4. new map from literals to set of suites
  for ( $\neg c$ : $C(p)$.getLiteralsExcept($p$))  | 4.5. for every literal in the $C(p)$,
  begin                                          | except literal $p$ itself, repeat:
                                                 | 4.5.1. if derivation path contains literal $c$
    if ( path.contains($c$) )                    | it is a “bad-loop” (to get $c$ we need $c$),
    begin                                        | literal $c$ is non-derivable for current path:
      $startClause$ = path.getStart($c$);        | – get clause in the derivation path,
                                                 | which starts the loop for the literal $c$
      c.setFinishedFor($startClause$);           | – set, that $c$ is finished for every
                                                 | path containing loop-starting clause
      continue;                                  | – continue with the next literal
                                                 | and go to step 4.5
    end                                          |
    $c\_suites$ = getSuitesFor($c$);             | 4.5.2. recursion call to get all suites for
                                                 | literal $c$ (including loop-suites)
    $suites.c$ -> $c\_suites$;                   | – add found suites into the suite map
    if ( path.contains($\neg c$) )               | 4.5.3. if derivation path contains $\neg c$
    begin                                        | it is a “good-loop”, $\neg c$ is derivable
                                                 | from the all clauses in the loop,
                                                 | if only all other literals are derivable
                                                 | (despite $p$ may be non-derivable):
      $loopSuite$ = new [; $loop$; $\neg$c];     | – create empty suite (without indices)
      $suites.c$.add($loopSuite$);               | – add suite into the suite map
    end                                          |
  end                                            |
  if ($suites$.containsForEveryLitaral())        | 4.6. if there is at least one suite
  begin                                          | for every literal of the clause $C(p)$:
    $p\_suites$ = genSuites($suites$, $C(p)$);   | – create new suites for the literal $p$
    $p$.suites.add($p\_suites$);                 | – add all found suites for the literal $p$
  end                                            | (obtained by using clause $C(p)$)
end                                              |
return $p$.suites;                               | 5. return found suites for the literal $p$
```

The following examples explain the main idea of the used loop-checking (good and bad loops) in the procedure.

Example 2 [Good-loop]. Suppose we have the set of clauses

$$S = \{D_{1.1} = p \vee \neg q \vee \neg z, D_{2.1} = q \vee \neg y, D_{3.1} = y \vee p, D_{4.1} = z \vee \neg p, D_{5.1} = \neg z \vee p\}$$

Clauses $p \vee \neg q \vee \neg z, q \vee \neg y, y \vee p$ creates “good-loop” for y, since for deriving p using $D_{1.1}$ it is enough to derive $q \vee p$ (and z): to derive p from the $p \vee \neg q \vee \neg z$, we need to derive q and z, to derive q, we need to derive y, to derive y, we need to derive $\neg p$. Since p is our goal, there is no need to eliminate p at all – we leave it in a loop suite.

Clauses $p \vee \neg q \vee \neg z, z \vee \neg p$ creates another “good-loop” for z.

Therefore, we create loop suites: $[3.1; loop; p]$ for y and, as a result, we create loop suite $[2.1, 3.1; loop; p]$ for q, and $[4.1; loop; p]$ for z.

Finally, from loop suites $[2.1, 3.1; \mathit{loop}; p]$ and $[4.1; \mathit{loop}; p]$ we generate closed loop $[1.1, 2.1, 3.1, 4.1; \mathit{closed};]$ for p.

Remark that q and z are non-derivable by themselves form the given set S.

Example 3 [Bad-loop]. Suppose we have the set of clauses

$$S = \{D_{1.1} = p \vee \neg q \vee \neg z, D_{2.1} = q \vee \neg y, D_{3.1} = y \vee \neg p, D_{4.1} = y \vee p, D_{5.1} = z\}$$

Clauses $p \vee \neg q \vee \neg z, q \vee \neg y, y \vee \neg p$ creates "bad-loop", since it is impossible to derive p using these clauses: to derive p from the $p \vee \neg q \vee \neg z$, we need to derive q and z, to derive q, we need to derive y, to derive y, we need to derive p. But p is our goal. Therefore, there is no sense to use such a derivation path for p derivation.

Despite this, it does not mean, that y is not derivable by itself using clause $D_{3.1} = y \vee \neg p$. $D_{3.1}$ is not useful for p derivation, but y may be derived from the $D_{3.1} = y \vee \neg p$ and $D_{4.1} = y \vee p$. This is the reason, we cannot mark y as non-derivable if "bad-loop" was found. We just mark it is explored for particular derivation path. Derivation procedure still derives y.

Note, that p is still derivable from the set $\{D_{1.1}, D_{2.1}, D_{4.1}, D_{5.1}\}$ in the given example, and PWATPL procedure returns suite $[1.1, 2.1, 4.1, 5.1; \mathit{closed}]$.

At first sight, it may look like PWATPL procedure is polynomial, since we explore every literal once. Unfortunately, it is not a true. "Bad loops" cause us to mark literals as explored for particular derivation path only. It means the same literal may be explored for the second time, and, therefore, we get an exponential complexity. It is not surprise, since it is 3SAT problem, which is known to be NP-complete according Cook theorem [2].

Lemma 1. *If Derivation procedure $getSuitesFor(p)$ for literal p returns suite $[i_1, i_2, i_3, \ldots, i_n; \mathit{loop/closed}; q_1, q_2, \ldots, q_k]$, then clause $p \vee q_1 \vee q_2 \vee \cdots \vee q_k$ is derivable from the set of clauses $S = \{D_{i_1}, D_{i_2}, \ldots, D_{i_n}\}$ using Resolution method.*

Proof. The proof goes from mathematical induction according suite size S size – n.

Lemma 2. *If clause $p \vee q_1 \vee q_2 \vee \cdots \vee q_k$ is derivable from the set of clauses $S = \{D_{i_1}, D_{i_2}, \ldots, D_{i_n}\}$, then suite $[i_1, i_2, i_3, \ldots, i_n; \mathit{loop/closed}; q_1, q_2, \ldots, q_k]$ will be returned by the Derivation procedure $getSuitesFor(p)$.*

Proof. The proof goes from mathematical induction according clause $p \vee q_1 \vee q_2 \vee \cdots \vee q_k$ derivation length m using Resolution method.

As the special case, we get, that literal p is derivable from the set of clauses $S = \{D_{i_1}, D_{i_2}, \ldots, D_{i_n}\}$ using Resolution method if and only if suite $[i_1, i_2, i_3, \ldots, i_n; \mathit{closed}]$ is returned by the Derivation procedure $getSuitesFor(p)$.

Therefore, PWATPL procedure is equivalent to classical Resolution method.

2 Conclusions

PWATPL procedure is a decision procedure with assumption traceability feature for propositional logic. Suites structures were used to "catch" loops, and 2 level indices were used to ensure getting smallest set of the used assumptions as a result. Despite PWATPL procedure returns much more information (set of needed and enough assumptions) comparing with classical derivation methods, its complexity is still as the Resolution method complexity for the worst case.

References

[1] R. Alonderis, H. Giedra, A. Pliuškevičienė, R. Pliuškevičius. Loop-type sequent calculi for temporal logic. *J. Autom. Reason.*, **64**:1663â€–1684, 2020.

[2] S.A. Cook. The complexity of theorem-proving procedures. In *Conference Record of Third Annual ACM Symposium on Theoryof Computing*, pp. 151–158, 1971.

[3] J. Dick. Rich traceability. In *Proc. 1st TEFSE*, pp. 18–23, 2002.

[4] G. Mints. Indexed systems of sequents and cut-elimination. *J. Philos. Log.*, **26**(6):671–696, 1997.

[5] S. Norgėla. *Matematinė logika.* TEV, Vilnius, 2004.

[6] A. Pliuškevičienė, R. Pliuškevičius. A new method to obtain termination in backward proof search for modal logic s4. *J. Log. Comput.*, **20**(1):353â€–379, 2010.

REZIUMĖ

Loginio išvedimo paieška su prielaidų atsekamumu

A. Birštunas, E. Reivytytė

Šiame darbe autoriai nagrinėja loginio išvedimo iš prielaidų atsekamumo problemą. Šio uždavinio esmė yra atsekti, kurios prielaidos iš turimos prielaidų žinių bazės yra būtinos tam tikros išvados išvedimui. Darbe pristatoma nauja teiginių logikai skirta išvedimo procedūra užtikrinanti prielaidų atsekamumo savybę. Išvedamai formulei išvedimo procedūra grąžina ir mažiausią prielaidų aibę, kurios užtenka išvados formulės išvedimui gauti. Procedūra verifikuota su autorių realizacija.

Raktiniai žodžiai: teiginių logika; atsekamumas; ciklų aptikimas