Legal Debugging in Propositional Legal Representation

Wachara Fungwacharakorn(✉) and Ken Satoh

National Institute of Informatics, Sokendai University, Tokyo, Japan
{wacharaf,ksatoh}@nii.ac.jp

Abstract. Literal interpretation on laws may produce unexpected consequences. They are difficult to be recognized unless exceptional cases were taken to the court. The court may decide a literal interpretation as exceptional, and then they have to identify which rule is a source of exception.

To assist the court, we proposed an idea called *legal debugging*, to find out which rule condition, called *a culprit*, causes unexpected consequences in such exceptional cases. We adapt the algorithmic program debugging with consideration of characteristics in reasoning in judgement, such as non-recursive stratified structures and factual propositions in order to find a culprit at last.

This paper presents legal debugging in propositional Prolog as well as PROLEG (PROlog based LEGal reasoning support system) specialized for legal reasoning. An example of legal debugging that interacts with a user and finds a culprit is also shown under the PROLEG representation of the case adapted from the real Supreme Court case.

Keywords: Legal reasoning · Legal representation · Algorithmic debugging

1 Introduction

Researchers have long been interested in representing legal knowledge in computers. Legal knowledge representation is usually divided into two types: rule-based and case-based. In rule-based legal representation, which is usually used for statutory laws, Legal rules are represented in logic programs such as Prolog [1, 2]. By formalizing the statute rules into computational logic, it provides benefits such as detecting conflicts in legal systems [3].

However, statute laws may be flawed. Even statute laws are cautiously drafted; some issues might be missing. These missing issues lead to unexpected consequences when we interpret the law literally. These issues are usually tacit, meaning that they are hard to know until such exceptional cases happen. When the cases are taken to the court, the court decided that the literal legal interpretation produces an unexpected consequence. The court has to identify which rule is a source of unexpected consequence and the court currently manually identifies such a rule so they might miss some sources of the unexpected result since exhaustive consideration might not be guaranteed.

Therefore, in this paper, we propose an idea of legal debugging, to find legal conditions that cause unexpected consequence, by imitating computer program debugging.

K. Kojima et al. (Eds.): JSAI-isAI 2018 Workshops, LNAI 11717, pp. 146–159, 2019.
https://doi.org/10.1007/978-3-030-31605-1_12

In contrast with legal conflicts that deal with other legal rules written explicitly, legal debugging has to deal with tacit expectations from the courts or the legal experts. This paper tries to identify the existence of unexpected consequences from those tacit expectations and propose legal debugging as a potential way to realize and solve unexpected consequences from laws.

In this paper, we report legal debugging based on algorithmic debugging, which originally proposed for tracing the difference between computation result from the program and expectation result from the user. Our technique traces the difference between a literal interpretation from the legal representation and an expected interpretation from legal experts. At this preliminary step of legal debugging development, we only focus on representation of statute laws which express the laws in written forms. This paper considers legal representations under propositional logic which is the basis of more advanced legal representations. We also take into account of characteristics of reasoning in judgements. For example, the logic rules should be stratified and not recursive so there would be only one interpretation which satisfies the rules.

This paper provides the means to find a legal bug, called a culprit, in two propositional based legal representations. The first representation in this paper is Prolog, which is more familiar to logic programmers. Although Prolog is not designed specifically for laws, a number of law formalizations have been tested in Prolog such as British Nationality Act [1] and the Income Tax Act of Canada [2]. The second representation this paper is PROLEG which stands for PROlog based LEGal reasoning support system [4]. PROLEG uses the concept of exception instead of negation which is more suitable to laws which usually separate between conditions and exceptions in legal documents. PROLEG was implemented based on the Presupposed Ultimate Fact Theory of Japanese Civil code, a legal reasoning scheme in real legal practice [5].

This paper is structured as follows. Section 2 describes propositional legal representation in logic program with negation as failure and defines a legal debugging process under the semantics. Section 3 extends the legal debugging process under the PROLEG semantics. Section 4 demonstrates a legal debugging under PROLEG using the Supreme Court case. Section 5 compares legal debugging and other related works on debugging. The final section summarizes the idea of legal debugging and its application in future works.

2 Legal Debugging Under Prolog

Prolog is a well-known logic programming based on a logic program with negation as failure. Generally, a logic program with negation as failure consists of rules in the form described as follows.

2.1 Basic Definitions

Definition 1. [Rule] A logic program with negation as failure is a finite set of rules Π which each element of Π is a rule R in the form $h \leftarrow b_1,\ldots,b_n, not\ c_1,\ldots, not\ c_m$ where h, b_i $(1 \leq i \leq n)$, and c_j $(1 \leq j \leq m)$ are propositions.

We denote h as $head(R)$, $\{b_1,...,b_n\}$ as $pos\_body(R)$, $\{c_1,...,c_m\}$ as $neg\_body(R)$, and $body(R)$ as $pos\_body(R) \cup neg\_body(R)$. We call a rule without a body a *fact*.

Definition 2. [Active rules] Let M be a set of propositions and Π be a logic program with negation as failure. A set of active rules of Π w.r.t. M denoted as Π^M is a set of $\{head(R) \leftarrow pos\_body(R) \mid \text{for all } R \in \Pi \text{ such that } neg\_body(R) \cap M = \varnothing\}$.

Definition 3. [Satisfaction] Let M be a set of propositions and R be a rule. M *satisfies* R if the following condition is satisfied: if $pos\_body(R) \subseteq M$ then $head(R) \in M$.

Definition 4. [Stable Model] Let Π be a logic program and M be a set of propositions. M is a stable model of Π if M is a minimum model of Π^M.

If M satisfies every rule in Π^M, M is a *model* of Π^M. The *minimum model* of Π^M is a model of Π^M which is the minimum in the sense of set inclusion.

Example 1. The following example shows a logic program with negation as failure P.

$p \leftarrow not\ q.$
$p \leftarrow not\ f.$
$q \leftarrow not\ r.$
$r \leftarrow f.$
$f \leftarrow .$

Let M be $\{f, r, p\}$, a set of active rules of P w.r.t. M or P^M is

$p \leftarrow .$
$r \leftarrow f.$
$f \leftarrow .$

Which M satisfies every rules in P^M and M is the minimum set that can do such things according to P^M. Therefore, M is a stable model of P.

Two assumptions from legal reasoning are introduced to distinguish the propositional legal representation from general logic programs. First assumption is that, generally, legal rules are not recursive. Recursive rules are those rules whose dependency graph contains a loop (see details in [6]). For example $\{p \leftarrow p.\}$, $\{p \leftarrow not\ p.\}$, $\{p \leftarrow q.\ q \leftarrow p.\}$ are recursive, but the program in Example 1 is not recursive. It is proved in [7] that a non-recursive program consists of only one stable model.

Definition 5. [Non-recursive program] A logic program Π is non-recursive when there is a partition $\Pi = \Pi_0 \cup \Pi_1 \cup \ldots \cup \Pi_n$ (Π_i and Π_j disjoint for all i $\neq j$) such that, for a predicate p appears in a body of rule in Π_i then a rule with p in the head is only contained within $\Pi_0 \cup \Pi_1 \cup \ldots \cup \Pi_j$ where $j < i$.

Second assumption is that some legal propositions are *factual*. Their truth values are usually evidence based so the court will finalize their truth values in a phase of fact finding and shall not reverse their truth values after that. Hence, such propositions are true only when given. From Example 1, f is factual because there is no rule that implies f (except a fact $f \leftarrow .$) so its truth value cannot be changed by altering other propositions. *factual* is defined as follows.

Definition 6. [Factual] A proposition α is *factual* w.r.t. a program Π if there is no rule that implies α except a fact $\alpha \leftarrow .$, in other words, α shall be true only when given.

However, some propositions could not be true concurrently. From Example 1, if r was true, q could not be true. We could say that $\{r\}$ (or any sets that contain r) does not support q. To determine this relation between rules and a set of propositions, the idea of *support* is defined as follows.

Definition 7. [Support] A set of propositions S *support*s a proposition α w.r.t. a logic program Π if there is a rule $R \in \Pi$ such that α is the head of the rule ($\alpha = head(R)$), $pos\_body(R) \subseteq S$, and $neg\_body(R) \cap S = \varnothing$. R is called a supporting rule of α w.r.t. S.

Theorem 1. [Relation between model and support] Given Π as a non-recursive logic and M is a stable model of Π, $\alpha \in M$ if and only if M supports α w.r.t. Π. (This relation is common. For further details, please see [8].

Proof

Suppose M supports α but $\alpha \notin M$, there is a supporting rule R of α in Π.

Hence $\alpha = head(R)$, $pos\_body(R) \subseteq M$ and $neg\_body(R) \cap M = \varnothing$.
But since $\alpha \notin M$, M does not satisfy $head(R) \leftarrow pos\_body(R)$ which exists in Π^M.
It leads to contradiction with the definition of model.

Suppose M does not support α but $\alpha \in M$, then no rules are supporting α.

Thus, M - $\{\alpha\}$ must be a model because it satisfies every rule in Π^M.
It leads to contradiction with the definition of the minimum model.

2.2 Formalizing Unexpected Consequences and Culprits

When the literal interpretation gives unexpected consequences, it means that we do not agree with the current stable model. However, the intended interpretation is not known explicitly in the first place but it rather reveals proposition by proposition when we consider the truth value of each proposition together with the debugger until we find a culprit defined to be a root cause of unexpected consequences.

However, because truth values of factual propositions are already finalized, the true factual proposition would not be possible to be excluded from the stable model and the false factual proposition would not be possible to be included in the stable model.

From Example 1, we could not intend $\{p, r\}$ to be a stable model because f is already given and we could not change its truth value. In contrast, we could intend $\{p, f\}$ to be the stable model because r is not factual so we allow changing the truth value of r. Therefore, we define an intended interpretation denoted as IM with following definitions.

Definition 8. [Intended interpretation] An intended interpretation IM of a non-recursive logic program Π is a set of propositions such that a set of factual propositions in the stable model of Π and a set of factual propositions in IM are equal.

Consequently, if IM is different from the stable model, there must be a non-factual α that is not currently derived but intended to be derived or currently derived but intended not to be derived. We call such propositions as *unexpected* which is defined as follows.

Definition 9. [Unexpected proposition] A proposition α is unexpected w.r.t. an intended interpretation IM and a non-recursive logic program Π with a stable model M if (1) $\alpha \notin M$ but $\alpha \in IM$ or (2) $\alpha \in M$ but $\alpha \notin IM$. Factual propositions could not be unexpected due to constraints in Definition 8.

To modify the program so its stable model would become what we intend, we must work on a non-factual proposition called *culprit* defined as follows.

Definition 10. [Culprit] a non-factual proposition α will be a *culprit* w.r.t. an intended interpretation IM and a logic program Π if

- $\alpha \in IM$ but IM does not support α w.r.t. Π or
- $\alpha \notin IM$ but IM supports α w.r.t. Π.

Then, we get the following theorem.

Theorem 2. [Finding a culprit] Given Π as a non-recursive logic program with a stable model M and IM as an intended interpretation that not equal to M. If α is unexpected w.r.t. IM and Π and α is not a culprit w.r.t. IM and Π, then there is another unexpected proposition $\beta \neq \alpha$ in a body of a rule that implies α.

Proof

If $\alpha \in IM$ and α is not a culprit

Then there is a rule R supporting α w.r.t. IM.
Hence, $pos\_body(R) \subseteq IM$ and $neg\_body(R) \cap IM = \emptyset$.
But $\alpha \notin M$ so R is not a supporting rule of α w.r.t. M.
Thus, $pos\_body(R) \not\subset min(\Pi)$ or $neg\_body(R) \cap min(\Pi) \neq \emptyset$.
Because Π is non-recursive, $\alpha \notin body(R)$
Hence, there is another proposition $\beta_1 \in pos\_body(R)$ such that $\beta_1 \in IM$ and $\beta_1 \notin M$ or another proposition $\beta_2 \in neg\_body(R)$ such that $\beta_2 \notin IM$ and $\beta_2 \in M$ that could be found in a body of a rule R that implies α.

If $\alpha \notin IM$ and $\alpha \in M$, there is a rule R supporting α w.r.t. M.

Hence, $pos\_body(R) \subseteq M$ and $neg\_body(R) \cap M = \emptyset$.
And if α is not a culprit, R is not a supporting rule of α w.r.t. IM.
Thus, $pos\_body(R) \not\subset IM$ or $neg\_body(R) \cap IM \neq \emptyset$.
Because Π is non-recursive, $\alpha \notin body(R)$
Hence, there is another proposition $\beta_1 \in pos\_body(R)$ such that $\beta_1 \notin IM$ and $\beta_1 \in M$ or another proposition $\beta_2 \in neg\_body(R)$ such that $\beta_2 \in IM$ and $\beta_2 \notin M$ that could be found in a body of a rule R that implies α.

From Example 1, the stable model is $\{p, r, f\}$. Let the intended interpretation IM be $\{q, f\}$, p, q, and r will become unexpected, and r is a culprit according to Theorem 2 (Table 1).

From Theorem 2, if we query from any unexpected proposition and find another unexpected proposition recursively, as long as the query sequence is finite and does not loop (e.g. a finite non-recursive logic program), the query finally succeeds by finding a culprit. Therefore, we could design the finding culprit algorithm as in Table 2.

Table 1. An illustrated example of Theorem 2

Rules	Unexpected found in a head	Supporting Rule w.r.t. the stable model	Supporting Rule w.r.t. *IM*	Unexpected found in a body
$p \leftarrow not\ q.$	p	Supporting	-	q
$p \leftarrow not\ f.$	p	-	-	-
$q \leftarrow not\ r.$	q	-	Supporting	r
$r \leftarrow f.$	r	Supporting	Supporting	- (thus r is a culprit w.r.t. *IM*)
$f.$	-	-	-	-

The algorithm is only for finding one culprit. In case there are two culprits or more, the user has to repeat the same procedure again. Actually, it is safe to check that the stable model from the revised program is equal to the intended interpretation. For example, from a program $\{p \leftarrow q,\ not\ r.\ q \leftarrow f_1.\ r \leftarrow f_2\}$ with an empty set as the stable model, if an intended interpretation was $\{q\}$, we can see that only q would be unexpected hence it would be a culprit. If we resolved by just adding q as a fact, $\{p, q\}$ would become a stable model, which is not same as what we intended. In another round, p would become unexpected w.r.t. $\{q\}$ and hence p would be a culprit. If we resolved by removing the rule $p \leftarrow q,\ not\ r$ from the program, $\{q\}$ would finally become the stable model as we expected.

Table 2. Finding culprit algorithm (a - They follow directly by the definition of a culprit, Definition 10. b - These conditions are actually determined when finding a supporting rule. They are mentioned to emphasize that they are unexpected.)

```
Input: a finite non-recursive logic program with a stable model
M, an intended interpretation IM, an unexpected proposition p
```

```
Find_culprit(p)
begin
  Find R as a supporting rule of p w.r.t. IM
  if p ∈ IM
    if[a] there is no such R return p;
    else
      Find q ∈ pos_body(R) s.t. q ∈ IM [b] and q ∉ M
      if there is such q return Find_culprit(q)
      Find r ∈ neg_body(R) s.t. r ∉ IM[b] and r ∈ M
      if there is such r return Find_culprit(r)

  if p ∉ IM
    if[a] there is such R return p;
    else
      Find R' as a supporting rule of p w.r.t. M
      Find q ∈ pos_body(R') s.t. q ∉ IM and q ∈ M[b]
      if there is such q return Find_culprit(q)
      Find r ∈ neg_body(R') s.t. r ∈ IM and r ∉ M [b]
      if there is such r return Find_culprit(r)
end
```

3 Legal Debugging Under PROLEG

PROLEG (PROlog based LEGal reasoning support system) [4] is a logic programming adapted from Prolog. It reflects the legal reasoning procedures called The Japanese Presupposed Ultimate Fact Theory practiced in Japanese law schools. PROLEG is different from Prolog in manipulation of negative conditions but the representation power of PROLEG is the same as Prolog [9]. In this section, we provide definitions for PROLEG and extend the legal debugging under PROLEG. First, these are basic definitions of PROLEG.

Definition 11. [PROLEG] A PROLEG program P is a pair $\langle H, E\rangle$ where

- H is a set of rules R of the from $h \leftarrow b_1,\ldots,b_n.$ where h and b_i $(1 \leq i \leq n)$ are propositions (note that there are no negations in the rule). We denote h as *head* (R) and $\{b_1,\ldots,b_n\}$ as *body*(R).
- E is a set of exceptions of the form *exception*(h, e) where h and e be propositions (note that e is a proposition, not a set of propositions).

Definition 12. [Applicable rule] Let M be a set of propositions and $\langle H, E\rangle$ be a PROLEG program. We denote *a set of applicable rules* w.r.t. M by $H^M = \{R \in H \mid \neg\exists exception(head(R), e) \in E$ such that $e \in M\}$.

So if an exception of rule exists in M $(e \in M)$ then the rule is inapplicable.

Definition 13. [Extension] A set of propositions M is an extension of a PROLEG program $\langle H, E\rangle$ if M is the minimum model of H^M $(M = \min(H^M))$.

Example 2. P' is a PROLEG program.

$$p \leftarrow q.$$
$$q \leftarrow f_1.$$
$$r \leftarrow f_2.$$
$$exception(p, r).$$

Let $M = \{r\}$, $H^M = \{q \leftarrow f_1,\ r \leftarrow f_2\ \}$ ($p \leftarrow q$ is inapplicable here because there is an *exception (p, r)* and $r \in M$). Since $\min(H^M) = \varnothing$. M is not an extension of P'.

Let $M = \varnothing$, $H^M = \{p \leftarrow q,\ q \leftarrow f_1,\ r \leftarrow f_2\ \}$. Since $\min(H^M) = \varnothing$. M is an extension of P'.

PROLEG representation is actually aligned with the logic program with negation as failure but using exception instead of negation. However, one particular different point is that if we add an exception to a condition, it applies to all rules on that condition unlike a logic program with negation as failure whose negations must be added to the rule one by one as illustrated in Table 3.

Because the representation power of PROLEG is same as the logic program with negation as failure, we can extend the same idea of supports, culprits, and finding culprit theorem by using an extension of P instead of the stable model. For example, these are definition of support (Definition 7), definition of intended interpretation (Definition 8), and definition of unexpected proposition (Definition 9) in PROLEG.

Table 3. An equivalent representation between PROLEG (left) and Prolog (right)

$p \leftarrow q$. $p \leftarrow r$. *exception*(p, e).	$p \leftarrow q$, *not* e. $p \leftarrow r$, *not* e.

Definition 14. [Support in PROLEG] A set of proposition S *supports* a proposition p w.r.t. a PROLEG program P if there is a rule R such that $p = head(R)$, $body(R) \subseteq S$, and there is no *exception(p, e)* $\in E$ such that $e \in S$ We call that R is a supporting rule of p w.r.t. S and P.

Definition 15. [Intended interpretation in PROLEG] A set of propositions IM can be an intended interpretation of a PROLEG program P if and only if a set of factual propositions in an extension of P and a set of factual propositions in IM are equal.

Definition 16. [Unexpected in PROLEG] A proposition p is unexpected w.r.t. an intended interpretation IM and a PROLEG program P if p is not in an extension of P but in IM or if p is in an extension of P but not in IM.

We design a finding culprit algorithm in PROLEG as shown in Table 4. It is still based on recursion from an unexpected proposition according to Theorem 2. Because the input PROLEG program P is not recursive, it can be deduced that there is only one extension of P.

Table 4. Finding culprit algorithm in PROLEG

Input: a finite non-recursive PROLEG logic program $P = \langle H, E \rangle$, an intended interpretation IM, an unexpected proposition p

```
Find_culprit(p)
begin
  Find R as a supporting rule of p w.r.t. IM
  if p ∈ IM
    if there is no such R return p;
    else
      Find q ∈ body(R) s.t. q is not in an extension of P
      if there is such q return Find_culprit(q)
      Find e s.t. exception(p,e) ∈ E and e is in an extension of P
      if there is such e return Find_culprit(e)
  if p ∉ IM
    if there is such R  return p;
    else
      Find R' as a supporting rule of p w.r.t. an extension of P
      Find q ∈ body(R') s.t. q ∉ IM
      if there is such q return Find_culprit(q)
      Find e s.t. exception(p,e) ∈ E and e ∈ IM
      if there is such e return Find_culprit(e)
end
```

4 Legal Debugging Example

In this section, we use an example of unexpected consequences adapted from this following case [10]:

1. A plaintiff made a lease contract for his house between him and the defendant.
2. When the defendant returned home for a while, he let his son use the room.
3. Then, the plaintiff claimed that the contract was ended by his cancellation for the reason that the defendant subleases without permission by literal interpretation of Japanese Civil Code Article 612 as follows.

> **Phrase 1:** A lessee may not assign the lessee's rights or sublease a leased thing without obtaining the approval of the lessor.
> **Phrase 2:** If the lessee allows any third party to make use of or take profits from a leased thing in violation of the provisions of the preceding paragraph, the lessor may cancel the contract.

When the case was taken to the court, the court decided that the literal interpretation produces an unexpected consequence. Although the cancellation is valid if we interpret the related piece of law literally, the court decided that the literal interpretation is too strict because "the third party" who makes use of the room temporally was the defendant's son and he did not harm the confidence between a lessee and a lessor, as the court mentioned the following:

> Phrase 2 is not applicable in exceptional situations where the sublease does not harm the confidence between a lessee and a lessor, and therefore the lessor cannot cancel the contract unless they prove the lessee's destructing of confidence.

The Japanese Civil Code Article 612 and the facts from the case can be represented in propositional PROLEG as in Table 5.

From this representation, `cancellation_due_to_sublease`, `effective_lease _contract`, and `effective_sublease_contract` are non-factual predicates in the extension of the program due to the given facts entailing these proposition and no exception is executed. This reflects when we interpret the law literally. However, since the court decided that the validity of `cancellation_due_to_sublease` is too harsh. It becomes an unexpected proposition. The legal debugger would help clarifying which legal conditions cause the unexpected consequence as well as finding the intended interpretation that supports the court reasoning. We could initiate debugging by using `cancellation_due_to_sublease` as an unexpected proposition as shown in Fig. 1.

The debugger firstly traced into the supporting rule of `cancellation_due_to_sublease` (the first rule) to determine two conditions in the body `effective_lease_contract` and `effective_sublease_contract`. The debugger asked user whether both conditions were intended to be fulfilled or not. If one of them was intended to be not fulfilled, it became a culprit because the intended interpretation would support it (situation 1 and 2). If both of them were intended to be fulfilled, the debugger retraced on `approval_of_sublease` which is an exception of `cancellation_due_to_sublease`. Then, the debugger asked user that

Table 5. Propositional PROLEG representation of Japanese Civil Code Article 612

```
cancellation_due_to_sublease <=
   effective_lease_contract,
   effective_sublease_contract,
   using_leased_thing,
   manifestation_cancellation.

effective_lease_contract <=
   agreement_of_lease_contract,
   handover_based_on_the_lease_contract.

effective_sublease_contract <=
   agreement_of_sublease_contract,
   handover_based_on_the_sublease_contract.

exception(cancellation_due_to_sublease,approval_of_sublease).


approval_of_sublease <=
  approval_of_sublease_before_the_day.
```

```
// Given Facts
   agreement_of_lease_contract.
   handover_based_on_the_lease_contract.
   agreement_of_sublease_contract.
   handover_based_on_the_sublease_contract.
   using_leased_thing.
   manifestation_cancellation.
   nonabuse_of_confidence.
```

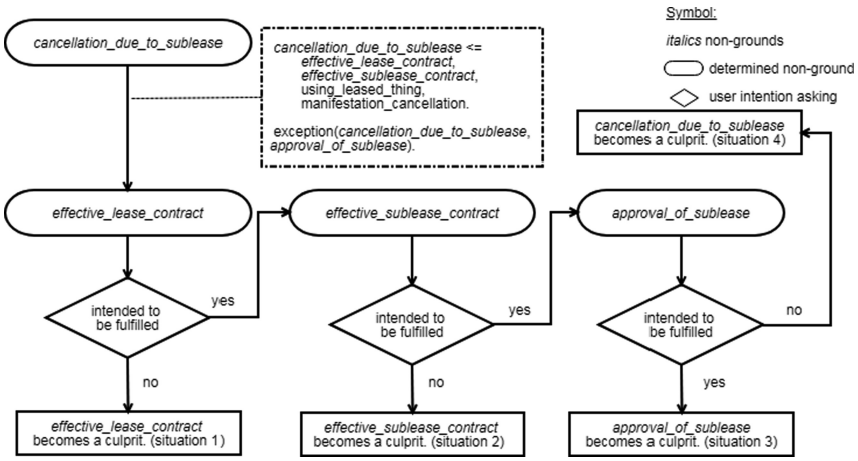

Fig. 1. Legal debugging steps from the rule base

approval_of_sublease was intended to be fulfilled or not. If it was intended to be fulfilled, it became a culprit because the intended interpretation would not support it (situation 3). If approval_of_sublease was intended to be not fulfilled, then there was no unexpected condition for cancellation_due_to_sublease, hence cancellation_due_to_sublease became a culprit itself (situation 4). The intended interpretations of each situation are illustrated in Table 6.

Table 6. Culprit and Intended Interpretation for Each Situation

Non-factual propositions in the current extension of the program: cancellation_due_to_sublease, effective_lease_contract, effective_sublease_contract		
Situation	Status of non-factual propositions given by the user (*IM* stands for an intended interpretation)	Found culprit
1	cancellation_due_to_sublease $\notin$ *IM* effective_lease_contract $\notin$ *IM*	effective_lease_contract (because it is not in *IM* but *IM* supports it w.r.t. the program)
2	cancellation_due_to_sublease $\notin$ *IM* effective_lease_contract $\in$ *IM* effective_sublease_contract $\notin$ *IM*	effective_sublease_contract (because it is not in *IM* but *IM* supports it w.r.t. the program)
3	cancellation_due_to_sublease $\notin$ *IM* effective_lease_contract $\in$ *IM* effective_sublease_contract $\in$ *IM* approval_of_sublease $\in$ *IM*	approval_of_sublease (because it is in *IM* but *IM* does not support it w.r.t. the program)
4	cancellation_due_to_sublease $\notin$ *IM* effective_lease_contract $\in$ *IM* effective_sublease_contract $\in$ *IM* approval_of_sublease $\notin$ *IM*	cancellation_due_to_sublease (because it is not in *IM* but *IM* supports it w.r.t. the program)

A culprit is considered in top-down left-to-right manner. Although the user does not consider all non-factual propositions, the debugger would return a first encountered culprit as soon as the debugger could not find any unexpected propositions. A culprit would be useful for considered rather in its resolution. Generally, the court would give an exception from extra facts of the case, such as in this case exception (cancellation_due_to_sublease, nonabuse_of_confidence) may be introduced to correspond to the Supreme Court decision. However, there are other possibilities to resolve one culprit so the resolution of culprits should be investigated further.

5 Discussion and Related Works

5.1 Legal Debugging in Statute Legal Practice

Statute rules are usually constructed in a top-down approach, from abstract to concrete concept. Each condition must be proved and presented to the court in the order of the list written in the procedure. For example, the case in Sect. 4 of this paper involves an

issue of cancellation of a lease contract due to sublease. To claim the issue, four conditions (effective lease contract, effective sublease contract must be effective, using the less thing, and manifesting cancellation) must be proved and presented to the court in order. However, when the court decided that the case produces an unexpected consequence. The court usually identifies the top concept to be unexpected. Therefore, the legal debugging helps the court to trace in a top-down manner from the abstract concept identified by the court to the culprit that causes unexpected consequences, as well as to trace in a left-to-right manner in order of the list written in the procedure.

5.2 Application of Debugging Besides Software

"Legal debugging" is proposed for tacit expectations unlike inconsistencies [11–14] that deals with conflicts between explicit written rules. Several paradigms have been proposed to find bugs such as online-offline justifications [15] and meta-programming [16] but most debugging technique are based on algorithmic debugging [17]. Besides software, algorithmic debugging has been proposed for navigating users in a few applications [18]. Zinn [19] has applied algorithmic debugging in tutoring systems. The papers view a program as a knowledge corpus and an intended interpretation as a student expectation. A student misconception can be viewed as a bug and a wrong answer can be viewed as an unexpected answer. Algorithmic debugging has also been applied in hardware design and verification. Kuchcinski et al. [20] has worked on using algorithmic debugging in hardware design by viewing circuits as auxiliary functions and logic programs respectively to detect faulty components.

Our paper is the first work proposing legal debugging. Legal debugging has to deal with tacit expectations from legal experts and different structures of representation, such as non-stratified structure and exception separation in PROLEG, and different resolution for preventing unexpected consequences, such as using exceptions instead of adding conditions. This paper views a representation of literal interpretation as a program and a *culprit*, a rule condition which causes unexpected consequences, as a bug.

5.3 Semantics of Legal Representation on Debugging

Program semantics may affect debugging schemes [18]. For example, in answer set programming, a debugger has to treat multiple situations due to the allowance of multiple answers [21–24]. In Datalog, a debugger has to deal with non-stratified programs differently because the semantics sometimes gives an empty set instead of non-termination for some types of non-stratified programs [25].

Since this paper is the first step on legal debugging, we have focused only a stratified and non-recursive representation. This representation often reflects the structure of statutory law that expects only one interpretation. Since a stratified and non-recursive program exists only one interpretation, we can eliminate the problems mentioned above. However, it is important to consider semantics used in legal representation because they still have some effects on legal debugging. For example, separation of conditions and exceptions in PROLEG affects resolution of legal culprits due to the border scope of exceptions.

6 Conclusion and Future Works

In this paper, we have proposed the idea of *legal debugging* in legal knowledge represented by logic programming. The idea has been presented in non-recursive program which we assume that some propositions' truth values shall not be changed (called *factual proposition*). Then, we have proposed the idea of *culprit*, a rule condition that causes unexpected consequences. We begin the debugging process from an initial *unexpected* proposition. The debugger follows a sequence of unexpected propositions until it meets a culprit otherwise the initial unexpected proposition is a culprit itself. We prove the correctness of algorithm under non-recursive logic programing with negation as failure, and then we extend the algorithm to PROLEG system. In future, we will extend the algorithm for first-order logic programs with arguments and develop an interactive debugger in PROLEG system which asks user intention and steps into rule base to find culprits similarly to computer program debugging.

Acknowledgement. We appreciate Randy Goebel, Oliver Ray, and Tiago Oliveira for their comments on the paper. This research is partially supported by JSPS KAKENHI Grant No. 17H06103.

References

1. Sergot, M.J., Sadri, F., Kowalski, R.A., Kriwaczek, F., Hammond, P., Cory, H.T.: The British Nationality Act as a logic program. Commun. ACM **29**, 370–386 (1986)
2. Sherman, D.M.: A Prolog model of the income tax act of Canada. In: Proceedings of the 1st International Conference on Artificial Intelligence and Law, pp. 127–136. ACM, New York (1987)
3. Li, T., Balke, T., De Vos, M., Satoh, K., Padget, J.: Detecting conflicts in legal systems. In: Motomura, Y., Butler, A., Bekki, D. (eds.) JSAI-isAI 2012. LNCS (LNAI), vol. 7856, pp. 174–189. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39931-2_13
4. Satoh, K., et al.: PROLEG: an implementation of the presupposed ultimate fact theory of Japanese civil code by PROLOG technology. In: Onada, T., Bekki, D., McCready, E. (eds.) JSAI-isAI 2010. LNCS (LNAI), vol. 6797, pp. 153–164. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-25655-4_14
5. Ito, S.: Lecture Series on Ultimate Facts. Shojihomu (2008). (in Japanese)
6. Ullman, J.: Principles of Database and Knowledge-Base Systems. Computer Science Press, Rockville (1988)
7. Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In: International Conference on Logic Programming/Joint International Conference and Symposium on Logic Programming, pp. 1070–1080 (1988)
8. Fages, F.: A new fixpoint semantics for general logic programs compared with the well-founded and the stable model semantics. New Gener. Comput. **9**, 425–443 (1991)
9. Satoh, K., Kogawa, T., Okada, N., Omori, K., Omura, S., Tsuchiya, K.: On generality of PROLEG knowledge representation. In: Proceedings of the 6th International Workshop on Juris-informatics (JURISIN 2012), Miyazaki, Japan, pp. 115–128 (2012)
10. Tokyo High Court: Case to seek removal of a building and surrender of lands. 1994 (O) 693. Minshu, vol. 50, no. 9 (1996)

11. Syrjänen, T.: Debugging inconsistent answer set programs. In: Proceedings of the 11th International Workshop on Nonmonotonic Reasoning, pp. 77–83 (2006)
12. Caminada, M., Sakama, C.: On the existence of answer sets in normal extended logic programs. In: Proceedings of the 2006 Conference on ECAI 2006: 17th European Conference on Artificial Intelligence, Riva Del Garda, Italy, pp. 743–744. IOS Press, Amsterdam (2006)
13. Schulz, C., Satoh, K., Toni, F.: Characterising and explaining inconsistency in logic programs. In: Calimeri, F., Ianni, G., Truszczynski, M. (eds.) LPNMR 2015. LNCS (LNAI), vol. 9345, pp. 467–479. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-23264-5_39
14. Ulbricht, M., Thimm, M., Brewka, G.: Measuring inconsistency in answer set programs. In: Michael, L., Kakas, A. (eds.) JELIA 2016. LNCS (LNAI), vol. 10021, pp. 577–583. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-48758-8_42
15. Pontelli, E., Son, T.C., Elkhatib, O.: Justifications for logic programs under answer set semantics. Theor. Pr. Log. Program. **9**, 1–56 (2009)
16. Gebser, M., Pührer, J., Schaub, T., Tompits, H.: A meta-programming technique for debugging answer-set programs. In: Proceedings of the 23rd National Conference on Artificial Intelligence – vol. 1, pp. 448–453. AAAI Press (2008)
17. Shapiro, E.Y.: Algorithmic Program Debugging. MIT Press, Cambridge (1983)
18. Caballero, R., Riesco, A., Silva, J.: A survey of algorithmic debugging. ACM Comput. Surv. **50**, 60:1–60:35 (2017)
19. Zinn, C.: Algorithmic debugging for intelligent tutoring: How to use multiple models *and* improve diagnosis. In: Timm, I.J., Thimm, M. (eds.) KI 2013. LNCS (LNAI), vol. 8077, pp. 272–283. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40942-4_24
20. Kuchcinski, K., Drabent, W., Maluszynski, J.: Automatic diagnosis of VLSI digital circuits using algorithmic debugging. In: Fritzson, P.A. (ed.) AADEBUG 1993. LNCS, vol. 749, pp. 350–367. Springer, Heidelberg (1993). https://doi.org/10.1007/BFb0019419
21. Fandinno, J., Schulz, C.: Answering the "why" in answer set programming – a survey of explanation approaches. Theor. Pract. Log. Program. **19**, 114–203 (2019)
22. Brain, M., De Vos, M.: Debugging logic programs under the answer set semantics. In: Answer Set Programming (2005)
23. Oetsch, J., Pührer, J., Tompits, H.: Catching the ouroboros: on debugging non-ground answer-set programs. Theor. Pract. Log. Program. **10**, 513–529 (2010)
24. Oetsch, J., Pührer, J., Tompits, H.: Stepping through an answer-set program. In: Delgrande, J.P., Faber, W. (eds.) LPNMR 2011. LNCS (LNAI), vol. 6645, pp. 134–147. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-20895-9_13
25. Caballero, R., García-Ruiz, Y., Sáenz-Pérez, F.: A theoretical framework for the declarative debugging of datalog programs. In: Schewe, K.D., Thalheim, B. (eds.) SDKB 2008. LNCS, vol. 4925, pp. 143–159. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-88594-8_8