Knowledge Representation – Prolog Systems

Seminar *Automated Reasoning*; TUM, Summer 2020

Johannes Neubrand

June 16, 2020

Abstract

This paper was written to complement the talk of the same name held as part of the Seminar *Automated Reasoning*.

It is intended to be an approachable introduction to logic programming and the SLD/SLDNF calculuses paralleling the approach of [4]. The Frame Problem is introduced and some approaches per [3] are discussed. The strengths of negation as failure are considered in light of these approaches.

Contents

Introduction

Prolog allows us to represent and reason about potentially unfinished sets of knowledge in an intuitive way. For example, consider someone who has read a few books about foreign sodas and learned about a particularly sweet brand, *Dr. Pepper*. They like sweet sodas in general, but hate nothing more than the taste of toothpaste. A Prolog formalization of these facts and rules is

```
sugary(dr_pepper).
likes(Soda) :- sugary(Soda), \+ toothpaste_undertone(Soda).
```

Here, `:-` means "is implied by", and `\+` should be read as "we cannot show".

Based on this knowledge, Prolog infers `likes(dr_pepper)`. When the subject finds out that Dr. Pepper reminds them of toothpaste and extends the program with `toothpaste_undertone(dr_pepper)`, the earlier statement is retracted.

To better understand the properties of this form of reasoning, we will first consider a restricted form thereof, named *SLD*.

SLD

SLD[1] is a simple calculus that allows some reasoning about atoms connected by rules.[2]

Rules describe logical relationships between literals. They correspond to tuples $(Head, Body)$, where $Head$ is a *literal* and $Body$ is a finite set of literals. They can be written $Head \leftarrow Body$, or simply $Head$ when $Body = \varnothing$. As in first-order logic, literals are optionally negated atoms. Some examples of rules are:

$$q; \quad p \leftarrow q, r; \quad r \leftarrow \neg s.$$

Read: q is true; p is true if both q and r are; r is true if s is false.

Programs are sets of rules over a set of atoms, $\mathbf{A}$. $Lit_{\mathbf{A}}$ signifies all literals corresponding to these atoms. A set of literals is called *consistent* iff it has no *complementary* literals (such as $a, \neg a$).

To make use of these simple programs, we are interested in finding the set of literals that are "consequences" of the rules. The consequences of a program Π are called $\mathrm{Cn}(\Pi)$.

In particular, we want such a set to be *closed* under Π; that is, for any rule $Head \leftarrow Body$ in Π where $Body \subseteq X$ holds, $Head \in X$ follows.

With this in mind, $\mathrm{Cn}(\Pi)$ is defined as the smallest set of literals over $\mathbf{A}$ which is both logically closed and closed under Π. The condition of logical closure enforces certain behavior when the program is inconsistent: for example, consider

$$a \leftarrow b; \quad \neg a \leftarrow b; \quad b.$$

It has the consequences $Lit_{\mathbf{A}}$ (including $\neg b$) since both $a, \neg a$ can be derived: as logical closure is required, all literals can be derived from this contradiction and

[1] Name stems from "Linear resolution with Selection function" for "Definite clauses", per [1]
[2] This section is condensed and adapted from [4].

thus must be elements of $\mathrm{Cn}(\Pi)$. In general, $\mathrm{Cn}(\Pi)$ is a consistent subset of $\mathit{Lit}_{\mathbf{A}}$ iff Π is consistent—otherwise, $\mathrm{Cn}(\Pi)$ is $\mathit{Lit}_{\mathbf{A}}$ itself.

Since we are interested only in literals (not formulas), the requirement of logical closure has no meaning outside of "exploding" when contradictions are derived.

The lack of a correspondence to first-order logic becomes obvious: $[\neg a; \quad a \leftarrow b]$ has the single consequence $\{\neg a\}$, while first-order logic may lead readers to try and (falsely!) apply the contrapositive to obtain $\neg a \rightarrow \neg b$ and therefore the incorrect set of consequences $\{\neg a, \neg b\}$. As [1] puts it, rules are "oriented".

Cn is a monotone function: $\Pi \subseteq \Pi' \Rightarrow \mathrm{Cn}(\Pi) \subseteq \mathrm{Cn}(\Pi')$. In other words, adding rules never decreases the set of consequences.

Even if a program has infinitely many rules, each of its consequences can be derived from a finite subset of the program.

A set of literals X is called *supported* by a program Π if for any literal $L \in X$, there is a rule $L \leftarrow \mathit{Body} \in \Pi$ such that $\mathit{Body} \subseteq X$. In particular, $\mathrm{Cn}(\Pi)$ is always supported by Π if Π is consistent.

A simple way to determine $\mathrm{Cn}(\Pi)$ is by finding the least fixed-point of T_Π, which is defined as follows:

$$T_\Pi(X) = \begin{cases} \{\mathit{Head} \mid \mathit{Head} \leftarrow \mathit{Body} \in \Pi, \mathit{Body} \subseteq X\} & X \text{ consistent} \\ \mathit{Lit}_{\mathbf{A}} & \text{otherwise} \end{cases}$$

$T_\Pi(X)$ is the set of literals that follow from X with regards to the rules from Π. In particular, for consistent X:

- $X \supseteq T_\Pi(X)$ iff X is closed.
- $X \subseteq T_\Pi(X)$ iff X is supported.

When $T_\Pi^n(\varnothing) = T_\Pi^{n+1}(\varnothing)$ (it stabilizes), the value $T_\Pi^n(\varnothing)$ is the least fixed point of T_Π. For finite programs, T_Π^n must always stabilize since T_Π is monotone.

SLD Calculus

The SLD calculus allows, to some extent, proofs of $L \overset{?}{\in} \mathrm{Cn}(\Pi)$. Its only axiom is $\models \varnothing$, and its inference rules are as follows:

$$\text{“Success” } (S)\ \frac{\models G \cup B}{\models G \cup \{L\}} \quad \text{for any } B \in \mathit{Bodies}(L)$$

$$\text{“Failure” } (F)\ \frac{=\!| \ G \cup B \quad \text{for all } B \in \mathit{Bodies}(L)}{=\!| \ G \cup \{L\}}$$

Here, the set of *bodies* of a literal is defined as follows:

$$\mathit{Bodies}(L) := \{\mathit{Body} \mid (L \leftarrow \mathit{Body}) \in \Pi\}$$

Intuitively expressed, a literal succeeds if some of its bodies can entirely be shown to succeed; meanwhile, a literal fails if all of its bodies can be shown to fail. A literal that has no bodies fails immediately.

For consistent programs, the success rule is both complete and sound[3], while the failure rule is incomplete, but nevertheless sound. In other words:

Use the (S) rule to show $L \in \mathrm{Cn}(\Pi)$ via $\models \{L\}$ (say: L *succeeds* relative to Π).

Use the (F) rule to show $L \notin \mathrm{Cn}(\Pi)$ via $=\!\mid \{L\}$ (say: L *fails* relative to Π), though it's possible $=\!\mid L$ cannot be shown for some non-consequences.

For instance, it is impossible to show $=\!\mid \{p\}$ wrt. the program $[p \leftarrow p]$.

As an example, consider the program [door_locked; ¬door_boarded_up; door_openable ← ¬door_locked, ¬door_boarded_up]. We can show:

$$\dfrac{(F)\ \dfrac{}{=\!\mid \{\neg\text{door\_locked}, \neg\text{door\_boarded\_up}\}}}{=\!\mid \{\text{door\_openable}\}}\ (F)$$

The upper derivation step holds since there is no body for ¬door_locked. Unfortunately, though door_openable fails, ¬door_openable fails too.

Closed-world Assumption

This shortcoming of SLD means that real-world programs written for it tend to be painfully verbose, containing many inference rules redundant under first-order logic. Instead of simply $[a \leftarrow b, c]$, logic programmers might want to write

$$a \leftarrow b, c; \quad \neg a \leftarrow \neg b; \quad \neg a \leftarrow \neg c.$$

The *closed-world assumption* states that, insofar as there is no *reason to believe a literal is true*, it should be *assumed* false.[4] After introducing the notation not to signify a literal that is not provable in this sense, one expresses the closed-world assumption as

$$L \leftarrow \mathsf{not}\,\overline{L} \quad \text{where } \overline{L} \text{ is the complementary literal of } L.$$

The semantics of not will be explored more thoroughly in the next section. For now, we will use Cn' to denote the adapted definition of consequence.

For example, consider the program

$$\text{door\_openable} \leftarrow \neg\text{door\_locked}.$$

Its consequences are $\varnothing$. After extending it with the closed-world assumption

$$\neg\text{door\_locked} \leftarrow \mathsf{not}\ \text{door\_locked}$$

to obtain Π', we can derive $\mathrm{Cn}'(\Pi') = \{\text{door\_openable}, \neg\text{door\_locked}\}$.

Note how expanding the program with

$$\text{door\_locked} \leftarrow \mathsf{not}\,\neg\text{door\_locked}$$

instead would have a markedly different effect, (the consequences would be {door_locked}): the closed-world assumption introduces an asymmetry between positive and negative literals.

In practice, the first form ($\neg A \leftarrow \mathsf{not}\,A$ for a positive literal A) finds more use.

[3]formally, completeness: $L \in \mathrm{Cn}(\Pi) \Rightarrow\ \models L$; soundness: $\models L \Rightarrow L \in \mathrm{Cn}(\Pi)$.

SLDNF

SLDNF is an extension of SLD that allows for *negation as failure* (NF).[4]

Rule elements are an extended notion of literals: they are optionally wrapped by the unary not operator. We extend programs to contain rule elements in their rules' bodies, and can rewrite any rule as $\mathit{Head} \leftarrow \mathit{Pos} \cup \mathsf{not}(\mathit{Neg})$. ($\mathsf{not}$ is applied pointwise on the set Neg.)

The intent is for not to refer to negation as failure: namely, $p \leftarrow \mathsf{not}\, q$ means that p can be derived if q cannot be.

Rules $\mathit{Head} \leftarrow \mathit{Pos} \cup \mathsf{not}(\mathit{Neg})$ are now applicable to a set of literals X whenever $\mathit{Pos} \subseteq X$ and $\mathit{Neg} \cap X = \varnothing$.

This changes the semantics of support: supported sets of literals are now those where each literal $L \in X$ is the head of a rule applicable to X.

Closure is redefined analogously: closed sets of literals are those X where the heads of all rules applicable to X are elements of X.

$$\begin{aligned}\text{Supported}(X) &:= \forall L \in X.\ \exists \mathit{Body} \in \mathcal{P}(X).\ (L \leftarrow \mathit{Body}) \in \Pi\\ \text{Closed}(X) &:= \forall (L \leftarrow \mathit{Body}) \in \Pi.\ \mathit{Body} \subseteq X \Rightarrow L \in X\end{aligned}$$

Unfortunately, these changes fundamentally break our previous definition of consequences, as well as our construction for Cn. Consider the program

$$p \leftarrow \mathsf{not}\, q; \quad q \leftarrow \mathsf{not}\, p.$$

In this instance, $T_\Pi^{2n+1}(\varnothing) = \{p, q\}$, while $T_\Pi^{2n}(\varnothing) = \varnothing$ for $n \in \mathbb{N}$ (when T is adjusted analogously to the definition of applicability).

In particular, the issue is that there no longer is a *smallest* set of literals which are both logically closed and closed under Π. In the previous example, $\{p\}, \{q\}$ are two such sets. Neither of them is smaller than the other with respect to $\subseteq$.

In order to formalize these kinds of sets of literals, the notion of *answer sets* is introduced. This is done by first defining the *reduct* operation Π^X

$$\Pi^X := \{L \leftarrow \mathit{Pos} \mid (L \leftarrow \mathit{Pos} \cup \mathsf{not}(\mathit{Neg})) \in \Pi, \mathit{Neg} \cap X = \varnothing\}.$$

A set X is an answer set if it satisfies $\text{Cn}(\Pi^X) = X$ as defined for SLD: X must be the smallest set of literals both logically closed and closed relative to the reduct Π^X.

Per this definition, all answer sets are logically closed. If they are consistent, they are additionally closed relative to Π and supported by Π (as redefined for SLDNF programs).

Simply requiring closure and supportedness of a set of literals does not suffice to characterize consistent answer sets. Consider $[p \leftarrow p]$. The set $\{p\}$ is both closed and supported while $\varnothing$ is the only answer set.

[4]This section is condensed and adapted from [4].

Additionally, $Lit_{\mathbf{A}}$ is an inconsistent answer set whenever the set of rules not using negation as failure is inconsistent as defined for SLD programs.

For programs that do not use not, there is exactly one answer set, which is the same as $\mathrm{Cn}(\Pi)$ as defined for SLD: $\Pi^X = \Pi$ for any such Π and arbitrary X.

Some programs, such as the example $[p \leftarrow \mathsf{not}\, q; \quad q \leftarrow \mathsf{not}\, p]$ discussed above, have multiple answer sets.

At last, Cn can be redefined as the intersection of all answer sets.

As previously hinted at, this definition is no longer monotone—in other words, $\Pi \subseteq \Pi' \not\Rightarrow \mathrm{Cn}(\Pi) \subseteq \mathrm{Cn}(\Pi')$. Consider the programs:

$$\Pi \equiv a \leftarrow \mathsf{not}\, b$$
$$\Pi' \equiv a \leftarrow \mathsf{not}\, b; \quad b \leftarrow \mathsf{not}\, a$$

Though $\Pi \subseteq \Pi'$, it also holds that $\mathrm{Cn}(\Pi) = \{a\} \nsubseteq \varnothing = \mathrm{Cn}(\Pi')$.

Literals belonging to all answer sets are called *well-founded.*
Literals belonging to no answer sets are called *unfounded.*

Closed-world Assumption in SLDNF

Given a consistent SLD program Π with *only positive head* literals, applying either operation:

- Π_1: for all $a \in \mathbf{A}$, replace literals of the form $\neg a$ with $\mathsf{not}\, a$;
- Π_2: for all $a \in \mathbf{A}$, add the rule $\neg a \leftarrow \mathsf{not}\, a$ to Π.

results in modified programs where $\mathrm{Cn}(\Pi_1) \cap \mathbf{A} = \mathrm{Cn}(\Pi_2) \cap \mathbf{A}$.

In other words, on a program without negation as failure that has positive head literals only, replacing logical negation with negation as failure is tantamount to adding the closed-world assumption.

SLDNF Calculus

The SLDNF calculus has the same axiom $\models \varnothing$ and these inference rules:

$$\text{“Success/Positive” } (SP)\ \frac{\models G \cup B}{\models G \cup \{L\}} \quad \text{for any } B \in \mathit{Bodies}(L)$$

$$\text{“Failure/Positive” } (FP)\ \frac{=\!\mid G \cup B \quad \text{for all } B \in \mathit{Bodies}(L)}{=\!\mid G \cup \{L\}}$$

$$\text{“Success/Negative” } (SN)\ \frac{\models G \qquad =\!\mid \{L\}}{\models G \cup \{\mathsf{not}\, L\}}$$

$$\text{“Failure/Negative” } (FN)\ \frac{\models \{L\}}{=\!\mid G \cup \{\mathsf{not}\, L\}}$$

The “Negative” pair of inference rules shows how the distinction $L, \mathsf{not}\, L$ translates rather directly to $\models, \Dashv$. In fact, the following correspondence holds:

$$\Dashv \{L\} \Leftrightarrow \models \{\mathsf{not}\ L\} \qquad \models \{L\} \Leftrightarrow \Dashv \{\mathsf{not}\ L\}$$

Soundness is expressed with respect to the previously introduced definition of well-founded and unfounded literals. Relative to a consistent program:

$$\models \{L\} \Rightarrow L \text{ is well-founded}; \quad \Dashv \{L\} \Rightarrow L \text{ is unfounded.}$$

SLDNF is not, in general, complete—even for success. For example, the program

$$a \leftarrow \mathsf{not}\, b; \quad b \leftarrow \mathsf{not}\, a; \quad c \leftarrow a; \quad c \leftarrow b$$

has the sole consequence $\{c\}$, but no derivation for $\models \{c\}$. For examples of SLDNF proof trees, please see the Appendix.

Prolog

Prolog systems are logic programming systems that allow for evaluation of goals with respect to a subset of SLDNF programs.[4]

In particular, Prolog systems generally stray from the previously introduced definition of SLDNF calculus by disallowing logical negation.

Prolog's syntax for SLDNF rules works according to this example:

SLDNF	Prolog
a	`a.`
$a \leftarrow b,\ \mathsf{not}\, c,\ d$	`a :- b, \+ c, d.`

As discussed earlier, using only negation as failure as in Prolog (disallowing all logical negation) is tantamount to using the closed-world assumption.

Prolog proofs generally allow the additional inference rule

$$\text{“Thinning”}\ (T)\ \frac{\Dashv G_1}{\Dashv G_1 \cup G_2}$$

A Simple Proof Strategy

A yet-to-be-proven or -disproven *goal* is denoted $\perp\!\!\!\perp G$. To prove some goal $\perp\!\!\!\perp G$:

- To process $\perp\!\!\!\perp \varnothing$, use the axiom to obtain $\models \varnothing$.
- To process $\perp\!\!\!\perp \{L\} \cup G$, attempt to prove $\perp\!\!\!\perp B \cup G \quad \forall B \in \mathit{Bodies}(L)$.
  - If $\forall B \in \mathit{Bodies}(L).\ \Dashv B \cup G$, use (FP) to derive $\Dashv G \cup \{L\}$.
  - If $\exists B \in \mathit{Bodies}(L).\ \models B \cup G$, use (SP) to derive $\models G \cup \{L\}$. (Discard any other bodies.)
- To process $\perp\!\!\!\perp \{\mathsf{not}\, L\} \cup G$, attempt to prove $\perp\!\!\!\perp L$.
  - If $\models L$, use (FN) to derive $\Dashv G \cup \{\mathsf{not}\, L\}$.
  - If $\Dashv L$, attempt to prove G.
    * If $\models G$, use (SN) to derive $\models G \cup \{\mathsf{not}\, L\}$.
    * If $\Dashv G$, use (T) to derive $\Dashv G \cup \{\mathsf{not}\, L\}$.

As can be seen from the list, the strategy will result in a definite answer $\dashv G$ or $\models G$ for any goal, assuming it terminates. Its results are sound.

However, it is possible that this proof strategy encounters infinite recursion, such as when attempting to prove $\perp\!\!\!\perp \{a\}$ on the program $[a \leftarrow a]$. By finding some requirement for programs that avoids this infinite recursion, one can show this strategy to be complete for consistent programs fulfilling it. SLDNF + Thinning and SLDNF themselves must also be complete for those programs.

One simple candidate is the existence of a measure on literals $\rho : \mathit{Lit}_{\mathbf{A}} \to \mathbb{N}$ s.t.

$$\forall(\mathit{Head} \leftarrow \mathit{Pos} \cup \mathsf{not}(\mathit{Neg})) \in \Pi.\ \forall L \in \mathit{Pos} \cup \mathit{Neg}.\ \rho(L) < \rho(\mathit{Head}).$$

Then, the depth of the call tree is finite. If there are finitely many rules, its breadth is finite on each level as well, and the call tree is finite overall.

A further extension of SLDNF generally found in Prolog systems allows for rules to contain *variables*. When this happens, the applicability of SLDNF's positive inference rules is based on whether the literal being operated on can be *unified* with some rule's head literal. Negative inference rules may be used only for variable-free literals.

```
crown_cap(paulaner_spezi).
screw_cap(adelholzener_limo).
hard_to_grip(Bottle) :- false.  % to avoid an existence_error
easy_to_open(Bottle) :- \+ crown_cap(Bottle),
                        \+ hard_to_grip(Bottle).
```

The third rule could be omitted in SLDNF, but Prolog requires all predicates and atoms to be the head of some rule. This avoids catastrophic results of typos.

To show `easy_to_open(adolholzener_limo)`, for instance, the first derivation step would lead to the subgoals `\+ crown_cap(adelholzener_limo)` and `\+ hard_to_grip(adelholzener_limo)`.

Note the use of nonmonotonicity: when `hard_to_grip(adelholzener_limo).` is added, `easy_to_open(adelholzener_limo)` is no longer a consequence.

The Frame Problem

In knowledge representation, the *frame problem* is the issue of describing which properties of some state can be changed by performing some action.[3] To illustrate this, we will consider an adapted version of the *Yale Shooting Scenario* based on [2].

In the following program, literals are either verbs ≈ *actions*, or adjectives ≈ properties of the *state*. The subscripts are intended as timestamps.

$$\begin{aligned}
&\mathrm{alive}_0\\
&\neg\mathrm{loaded}_0\\
&\mathrm{load}_0\\
&\mathrm{loaded}_1 \leftarrow \mathrm{load}_0\\
&\mathrm{shoot}_2\\
&\neg\mathrm{alive}_3 \leftarrow \mathrm{loaded}_2,\ \mathrm{shoot}_2
\end{aligned}$$

(Though beyond the scope of this paper, some expressions of the frame problem use a *state* $\times$ *action* $\rightarrow$ *state* transition function in lieu of linear time.)

The intended meaning of the program is that some being is alive initially (alive_0), while some gun is unloaded initially ($\neg\text{loaded}_0$). Then, our gunslinger loads the gun, waits a bit, and finally pulls the trigger. Since they have impeccable aim, they always hit the target, which unfortunately dies shortly after.

The fourth and sixth rules describe actions whose execution is required by the third and fifth rules, respectively.

The issue is, how do we guarantee that exactly the intended properties are changed by these actions? Ideally, one would like to assume that properties are unchanged insofar as they are not affected by any action.

Adding rules akin to $\text{loaded}_1 \leftarrow \text{loaded}_0$ may seem like an option, but they often prevent a too broad set of changes to the state. With this in mind, what more general approaches make sense?

Approaches

Explanation Closure

When looking beyond logic programming, one could take an approach known as *explanation closure*: whenever any change to the state happens, require that it is adequately explained.[3] In first-order logic, this could manifest itself as axioms

$$\begin{aligned}
\neg\text{loaded}_t \land \quad \text{loaded}_{t+1} &\rightarrow \text{load}_t \\
\text{loaded}_t \land \neg\text{loaded}_{t+1} &\rightarrow \bot \\
\neg\text{alive}_t \land \quad \text{alive}_{t+1} &\rightarrow \bot \\
\text{alive}_t \land \neg\text{alive}_{t+1} &\rightarrow \text{shoot}_t \land \text{loaded}_t.
\end{aligned}$$

for arbitrary t. This approach is obviously heavily verbose, but it allows for a clean definition of the semantics of actions. Crucially, it does not make direct statements about when the state remains *the same*, instead only ensuring that every *change* has an adequate explanation.

Though this approach to the frame problem is correct, it is "backwards" in a sense: a direct translation of explanation closure statements to logic programming yields rules that infer information about an "earlier" state from a "later" one. *No statements about the "future" can be made* since p_{t+1} is not a head literal.

Default Calculus

To avoid explanation closure's shortcomings, some have taken the approach of taking the intent of solutions to the frame problem: "properties remain the same unless acted upon." This was rephrased it as "properties remain the same when there is no evidence to the contrary," and directly using that assumption.[5]

For instance: "If the gun is loaded at some time t, and if there is no evidence that it is *un*loaded at time $t + 1$, assume that it is loaded at time $t + 1$ as well."

The *default calculus* as introduced by Reiter [5] allows formalizing such relationships. In it, theories can contain *defaults* in addition to first-order axioms.

$$\frac{P \;:\; \mathsf{M}\, Q_1 \cdots}{R}\,.$$

stands for a default relating first-order formulas $P, Q_1, \ldots$, and R. The intended meaning is "Given P; R follows if $Q_1, \ldots$ are consistent with *everything else*."

Of course, *everything else* is hard to put into words, and risks introducing circularity. There is nevertheless a very pragmatic notion of an *extension*: a set of formulas closed under first-order derivation with the axioms and applicable defaults.

The set of formulas *entailed* by the theory is the intersection of all extensions. (This is, of course, heavily reminiscent of the relationship between SLDNF consequences and answer sets, though the sets themselves differ heavily: $Lit_{\mathbf{A}}$, which can be finite, versus arbitrarily large first-order formulas.)

A solution to the frame problem named the *frame default* is described by

$$\frac{p_t \;:\; \mathsf{M}\, p_{t+1}}{p_{t+1}} \qquad \text{for all properties } p.$$

Abnormality Logic

The final approach treated by [3] is *abnormality.* As used in [2], abnormality predicates can be used to formalize under which conditions some *state propagation rule* should *not* apply.

Generally, the intuition is that $\mathrm{Ab}_{s,a,t}$ should be true if the property s could be affected by taking action a. Its precise structure depends on the action itself.

These definitions would suffice for our modified Yale Shooting Problem:

$$\mathrm{Ab}_{\mathrm{alive;shoot;}t} \leftarrow \mathrm{loaded}_t \qquad \mathrm{Ab}_{\mathrm{loaded;load;}t} \leftarrow \neg\mathrm{loaded}_t$$

Rule 1 could be read as: "Given that $loaded_t$, the property $alive_{t+1}$ will behave abnormally if action $shoot_t$ is taken". The closed world assumption is then added (if we are not using NF).

The following state propagation rule is usually used:

$$\sim\! x_{t+1} \leftarrow \neg\mathrm{Ab}_{x;a;t},\, a_t,\, \sim\! x_t \qquad \begin{array}{l}\text{where } a \text{ is an action,}\\ x \text{ is a property,}\\ \sim\! x \in \{x, \neg x\}.\end{array}$$

Prolog and Default Logic

As hinted at earlier, SLDNF rules like $Head \leftarrow Pos \cup \mathsf{not}(Neg)$ correspond roughly to defaults of the form

$$\frac{\bigwedge Pos \;:\; \mathsf{M}\, \overline{Neg}}{Head}$$

where $\overline{Neg}$ is the pointwise complement of the literals in Neg. (In Prolog, literals can only be positive, so $\overline{Neg}$ contains only negative literals.)

However, it is entirely possible to create new literals that represent these logically negated literals. They can be made mutually exclusive in a rather strong sense: $[a \leftarrow \mathsf{not}\, a'; \quad a' \leftarrow \mathsf{not}\, a]$, for example, ensures that a, a' are never both elements of a supported set, and in particular, of a consistent answer set.

Nonmonotonicity in Practice

As mentioned earlier, the consequence function Cn is not monotone in SLDNF. In particular, adding more rules can decrease the set of derivable literals.

This fact may have seemed absurd initially. However, given the presented approaches to the frame problem, it seems useful to allow for "beliefs" that can be retracted as more information is available. The nonmonotonicity afforded to us by SLDNF is uniquely useful in practice and allows application of both the frame default and abnormality logic.

Abnormality logic in Prolog is implemented by replacing the state-propagation rule's use of $\neg\mathrm{Ab}$ with $\mathsf{not}\,\mathrm{Ab}$.

$$\begin{aligned} &\mathrm{Ab}_{\mathrm{alive;shoot;t}} \leftarrow \mathrm{loaded}_t \\ &\mathrm{alive}_{t+1} \leftarrow \mathsf{not}\,\mathrm{Ab}_{\mathrm{alive};a;t},\ a_t,\ \mathrm{alive}_t \qquad \text{for all actions } a \end{aligned}$$

Reasoning with the frame default generally requires the usage of pseudo-complementary literals. Our example's propagation rules would turn into

$$\begin{aligned} \mathrm{loaded}_{t+1} &\leftarrow \quad \mathrm{loaded}_t,\ \mathsf{not}\,\mathrm{unloaded}_{t+1} \\ \mathrm{unloaded}_{t+1} &\leftarrow \mathrm{unloaded}_t,\ \mathsf{not} \quad \mathrm{loaded}_{t+1} \end{aligned}$$

Assuming the program is consistent, this ensures that at most one of the literals loaded_t and $\mathrm{unloaded}_t$ is a consequence of the program at any t.

Both of these examples are explored further in the first part of the Appendix.

Even logic programs that reason in paradigms not based on stateful logic can benefit from an expanded notion of abnormality:

```prolog
sugary(paulaner_spezi).
sugary(adelholzener_limo).
insect_took_a_bath_in(adelholzener_limo).
ab(Beverage) :- insect_took_a_bath_in(Beverage).
tasty(Beverage) :- sugary(Beverage), \+ ab(Beverage).
```

Conclusion

Those interested in the implementation of variables in Prolog should take a look at Lifschitz's survey [4]. The paper also includes more information about how they relate Prolog to certain database systems, as well as a short exposé on answer set solvers that avoid some pitfalls of SLDNF-based solution mechanisms over logic programs with negation with failure.

For a more thorough discussion of default logic beyond negation as failure, as well as some examples that admittedly have not aged too well, consider reading Reiter's article [5].

References

[1] Gerhard Brewka and Jürgen Dix. “Knowledge Representation with Logic Programs.” In: *Lecture Notes in Computer Science 1471: Logic Programming and Knowledge Representation*. Springer, 1997, pp. 1–51.

[2] Steve Hanks and Drew McDermott. “Default Reasoning, Nonmonotonic Logics, and the Frame Problem.” In: *AAAI-86 Proocedings*. AAAI, 1986, pp. 328–333.

[3] Vladimir Lifschitz. “The Dramatic True Story of the Frame Default.” In: *Journal of Philosophical Logic* (44 2015), pp. 163–176.

[4] Vladimir Lifschitz and Hudson Turner. “Foundations of Logic Programming.” In: *Principles of Knowledge Representation*. CSLI Publications, 1996, pp. 69–127.

[5] Raymond Reiter. “A Logic for Default Reasoning.” In: *Artificial Intelligence* (13 1980), pp. 81–132.

Appendix

As a more realistic example of SLDNF Calculus, we will discuss this logic program Π that describes whether a specific person has correctly paid their taxes:

$$\begin{aligned}
&\text{taxes\_late} \leftarrow \texttt{not}\ \text{paid}, \text{past\_june}\\
&\text{problematic} \leftarrow \text{taxes\_late}, \texttt{not}\ \text{extension\_requested}\\
&\text{problematic} \leftarrow \text{lied}
\end{aligned}$$

Or, in Prolog,

```
taxes_late :- \+ paid, past_june.
problematic :- taxes_late, \+ extension_requested.
problematic :- lied.
```

This program fulfills the requirements outlined in the section “A Simple Proof Strategy” for programs with a strictly monotonically decreasing measure on literals. Because of this, we can expect the proof search to terminate in all cases.

In the following, we will show proof trees for $\perp\!\!\!\perp \{\mathit{problematic}\}$ for various different programs based on this one.

First, let’s consider someone who has paid their taxes with nothing amiss.

$$\Pi \cup \{\text{paid}, \text{past\_june}\}$$

$$\dfrac{(FP)\;\dfrac{(FN)\;\dfrac{(SP)\;\dfrac{\models \varnothing}{\models \{\text{paid}\}}}{\Dashv \{\texttt{not}\ \text{paid}, \text{past\_june}, \texttt{not}\ \text{extension\_requested}\}}}{\Dashv \{\text{taxes\_late}, \texttt{not}\ \text{extension\_requested}\}} \qquad (FP)\;\dfrac{}{\Dashv \{\text{lied}\}}}{\Dashv \{\text{problematic}\}}\;(FP)$$

Note that removing *past_june* has no effect on the outer form of the search tree using the given search strategy.

Now, let's model a tax evader who already paid (an incorrect sum).

$$\Pi \cup \{\text{paid}, \text{lied}\}$$

$$(SP)\dfrac{(FP)\dfrac{(FN)\dfrac{(SP)\dfrac{\models \varnothing}{\models \{\text{paid}\}}}{\Dashv \{\textsf{not}\ \text{paid}, \text{past\_june}, \textsf{not}\ \text{extension\_requested}\}}}{\Dashv \{\text{taxes\_late}, \textsf{not}\ \text{extension\_requested}\}} \qquad (SP)\dfrac{\models \varnothing}{\models \{\text{lied}\}}}{\models \{\text{problematic}\}}$$

The gray left-hand-side is not actually used for the (SP) inference rule, and so this proof usually would not contain it. It's shown here to build an intuition about the depth-first nature of Prolog's search strategy outlined earlier.

Next, let's examine someone who procrastinated for a bit too long.

$$\Pi \cup \{\text{past\_june}\}$$

$$(SP)\dfrac{(SP)\dfrac{(SN)\dfrac{(FP)\dfrac{}{\Dashv \{\text{paid}\}} \qquad (SP)\dfrac{(SN)\dfrac{(FP)\dfrac{}{\Dashv \{\text{extension\_requested}\}} \qquad \models \varnothing}{\models \{\textsf{not}\ \text{extension\_requested}\}}}{\models \{\text{past\_june}, \textsf{not}\ \text{extension\_requested}\}}}{\models \{\textsf{not}\ \text{paid}, \text{past\_june}, \textsf{not}\ \text{extension\_requested}\}}}{\models \{\text{taxes\_late}, \textsf{not}\ \text{extension\_requested}\}}}{\models \{\text{problematic}\}}$$

This time, the search strategy saved time since the *lied*-subtree was skipped.

Had this person requested an extension, the tree would instead have been:

$$\Pi \cup \{\text{extension\_requested}, \text{past\_june}\}$$

$$(FP)\dfrac{(FP)\dfrac{(T)\dfrac{(FP)\dfrac{}{\Dashv \{\text{paid}\}} \qquad (FP)\dfrac{(FN)\dfrac{(SP)\dfrac{\models \varnothing}{\models \{\text{extension\_requested}\}}}{\Dashv \{\textsf{not}\ \text{extension\_requested}\}}}{\Dashv \{\text{past\_june}, \textsf{not}\ \text{extension\_requested}\}}}{\Dashv \{\textsf{not}\ \text{paid}, \text{past\_june}, \textsf{not}\ \text{extension\_requested}\}}}{\Dashv \{\text{taxes\_late}, \textsf{not}\ \text{extension\_requested}\}} \qquad (FP)\dfrac{}{\Dashv \{\text{lied}\}}}{\Dashv \{\text{problematic}\}}$$

Again, the gray portion is unused in the final proof tree but generated nevertheless by the proof search.

We can also see some evidence that the Thinning rule does not strengthen SLDNF. By rearranging the goal items, we can avoid it and simplify the tree:

$$\Pi \cup \{\text{extension\_requested}, \text{past\_june}\}$$

$$(FP)\dfrac{(FN)\dfrac{(SP)\dfrac{\models \varnothing}{\models \{\text{extension\_requested}\}}}{\Dashv \{\textsf{not}\ \text{extension\_requested}, \text{taxes\_late}\}} \qquad (FP)\dfrac{}{\Dashv \{\text{lied}\}}}{\Dashv \{\text{problematic}\}}$$

Let's now consider the Yale Shooting Problem as described earlier, but reduced to the final state change. (We now index the previously-second timestamp as 0. Additionally, we no longer apply the full Prolog search process for brevity.)

First, characterizing it using abnormality logic yields:

$$\begin{aligned}
\mathrm{ab}_{\mathrm{alive;shoot;0}} &\leftarrow \mathrm{loaded}_0\\
\mathrm{ab}_{\mathrm{loaded;load;0}} &\leftarrow \mathsf{not}\ \mathrm{loaded}_0\\
\mathrm{loaded}_1 &\leftarrow \mathrm{loaded}_0, \mathrm{load}_0, \mathsf{not}\ \mathrm{ab}_{\mathrm{loaded;load;0}}\\
\mathrm{alive}_1 &\leftarrow \mathrm{alive}_0, \mathrm{load}_0, \mathsf{not}\ \mathrm{ab}_{\mathrm{alive;load;0}}\\
\mathrm{loaded}_1 &\leftarrow \mathrm{loaded}_0, \mathrm{shoot}_0, \mathsf{not}\ \mathrm{ab}_{\mathrm{loaded;shoot;0}}\\
\mathrm{alive}_1 &\leftarrow \mathrm{alive}_0, \mathrm{shoot}_0, \mathsf{not}\ \mathrm{ab}_{\mathrm{alive;shoot;0}}\\
\mathrm{loaded}_1 &\leftarrow \mathrm{load}_0
\end{aligned}$$

$$\mathrm{alive}_0;\quad \mathrm{loaded}_0;\quad \mathrm{shoot}_0$$

To show $\models \{\mathit{loaded}_1\}$ and $\Dashv \{\mathit{alive}_1\}$:

$$(SP)\;\dfrac{(SP)\;\dfrac{(SP)\;\dfrac{(SN)\;\dfrac{\models \varnothing \qquad (FP)\;\dfrac{}{\Dashv \{\mathrm{ab}_{\mathrm{loaded;shoot;0}}\}}}{\models \{\mathsf{not}\ \mathrm{ab}_{\mathrm{loaded;shoot;0}}\}}}{\models \{\mathrm{shoot}_0, \mathsf{not}\ \mathrm{ab}_{\mathrm{loaded;shoot;0}}\}}}{\models \{\mathrm{loaded}_0, \mathrm{shoot}_0, \mathsf{not}\ \mathrm{ab}_{\mathrm{loaded;shoot;0}}\}}}{\models \{\mathrm{loaded}_1\}}$$

$$(FP)\;\dfrac{(FN)\;\dfrac{(SP)\;\dfrac{(SP)\;\dfrac{\models \varnothing}{\models \{\mathrm{loaded}_0\}}}{\models \{\mathrm{ab}_{\mathrm{alive;shoot;0}}\}}}{\Dashv \{\mathrm{alive}_0, \mathrm{shoot}_0, \mathsf{not}\ \mathrm{ab}_{\mathrm{alive;shoot;0}}\}} \qquad (FP)\;\dfrac{}{\Dashv \{\mathrm{alive}_0, \mathrm{load}_0, \mathsf{not}\ \mathrm{ab}_{\mathrm{alive;load;0}}\}}}{\Dashv \{\mathrm{alive}_1\}}$$

Finally, restating this part of the Yale Shooting Problem with the frame default:[5]

$$\begin{aligned}
\mathrm{unloaded}_1 &\leftarrow \mathrm{unloaded}_0, \mathsf{not}\ \mathrm{loaded}_1\\
\mathrm{loaded}_1 &\leftarrow \mathrm{loaded}_0, \mathsf{not}\ \mathrm{unloaded}_1\\
\mathrm{dead}_1 &\leftarrow \mathrm{dead}_0, \mathsf{not}\ \mathrm{alive}_1\\
\mathrm{alive}_1 &\leftarrow \mathrm{alive}_0, \mathsf{not}\ \mathrm{dead}_1\\
\mathrm{dead}_1 &\leftarrow \mathrm{shoot}_0, \mathrm{loaded}_0\\
\mathrm{loaded}_1 &\leftarrow \mathrm{load}_0
\end{aligned}$$

$$\mathrm{alive}_0;\quad \mathrm{loaded}_0;\quad \mathrm{shoot}_0$$

Assuming the past is pseudo-consistent[6], the future is kept pseudo-consistent by default rules. We can show (roots of interesting subtrees emphasized):

$$(SP)\;\dfrac{(SP)\;\dfrac{(SN)\;\dfrac{\models \varnothing \qquad (FP)\;\dfrac{(FP)\;\dfrac{}{\Dashv \{\mathrm{unloaded}_0, \mathsf{not}\ \mathrm{loaded}_1\}}}{\Dashv \{\mathrm{unloaded}_1\}}}{\models \{\mathsf{not}\ \mathrm{unloaded}_1\}}}{\models \{\mathrm{loaded}_0, \mathsf{not}\ \mathrm{unloaded}_1\}}}{\models \{\mathrm{loaded}_1\}}
\qquad
(FP)\;\dfrac{(FN)\;\dfrac{(SP)\;\dfrac{(SP)\;\dfrac{(SP)\;\dfrac{\models \varnothing}{\models \{\mathrm{shoot}_0\}}}{\models \{\mathrm{shoot}_0, \mathrm{loaded}_0\}}}{\models \{\mathrm{dead}_1\}}}{\Dashv \{\mathrm{alive}_0, \mathsf{not}\ \mathrm{dead}_1\}}}{\Dashv \{\mathrm{alive}_1\}}$$

[5]Ordering (*of* rules and *within* rules) is far more important than in the previous program, as we could accidentally introduce cyclical rules that cause infinite recursion to occur.

[6]In other words, a literal and its pseudo-complement may never both hold.