

Automated Defeasible Reasoning in Law

NN1¹, Avishkar Mahajan^{2[nr]}, Liyana Muthalib^{2[nr]}, and Martin Strecker^{3[0000-0001-9953-9871]}

¹ Foo, SG

² Bar, SG

³ Baz, SG

Abstract.

Keywords: Knowledge representation and reasoning, Argumentation and law, Computational Law, Defeasible reasoning

1 Introduction

Computer-supported reasoning about law is a longstanding effort of researchers from different disciplines such as jurisprudence, artificial intelligence, logic and philosophy. What originally may have appeared as an academic playground is now evolving into a realistic scenario, for various reasons.

On the *demand* side, there is a growing number of human-machine or machine-machine interactions where compliance with legal norms is essential, such as in sales, insurance, banking and finance or digital rights management, to name but a few. Innumerable “smart contract” languages attest of the interest to automate these processes, even though many of them are rather dedicated programming languages than formalisms intended to express and reason about regulations.

On the *supply* side, decisive advances have been made in fields such as automated reasoning and language technologies, both for computerised domain specific languages (DSLs) and natural languages. Even though a completely automated processing of traditional law texts capturing the subtleties of natural language is currently out of scope, one can expect to code a law text in a DSL that is amenable to further processing.

This “rules as code” approach is the working hypothesis of our CCLAW project⁴: law texts are formalised in a DSL called L4 that is sufficiently precise to avoid ambiguities of natural languages and at the same time sufficiently close to a traditional law text with its characteristic elements such as cross references, prioritisation of rules and defeasible reasoning. Indeed, presenting these features is one of the main topics of this paper. Once a law has been coded in L4, it can be further processed: it can be converted to natural language [?] to be as human readable as a traditional law text, and efficient executable code can be extracted, for example to perform tax calculations (all this is not the topic

⁴ <https://cclaw.smu.edu.sg/>

of the present paper). It can also be analysed, to find faults in the law text on the meta level (such as consistency and completeness of a rule set), but also on the object level, to decide individual cases.

Overview of the paper The main emphasis of this paper is on the L4 DSL that is currently under definition, which in particular features a formalism for transcribing rules and reasoning support for verifying their properties. Section ?? is dedicated to a description of the language and the L4 system implementation currently under development. The rule language will be dissected in Section ?. We will in particular describe mechanisms for prioritisation and defeasibility of rules that are encoded via specific keywords in law texts. We then define a precise semantics of these mechanisms, by a translation to logic. Classical, monotonic logic, developed in Section ?, has received surprisingly little attention in this area, even though proof support in the form of SAT/SMT solvers has made astounding progress in recent years. An alternative approach, based on Answer Set Programming, is described in Section ?. We conclude in Section ?.

Related work ctd

MS: TBD

- For bibliographical references, see ⁵
- [?,?]
- [?]
- [?]

2 An Overview of the L4 Language

This section gives an account of the L4 language as it is currently defined – as an experimental language, L4 will evolve over the next months. In our discussion, we will ignore some features such as a natural language interface [] which is not relevant for the topic of this paper but may be [desirable/explored/necessary in the future for some reason].

As a language intended for representing legal texts and reasoning about them, an L4 module is essentially composed of four sections:

- a terminology in the form of *class definitions*;
- *declarations* of functions and predicates;
- *rules* representing the core of a law text, specifying what is considered as legal behaviour;
- *assertions* for stating and proving properties about the rules.

In the following, these elements will be presented in more detail and illustrated with an example, a (fictitious) regulation of speed limits for different types of vehicles.

⁵ <https://law.stanford.edu/publications/developing-a-legal-specification-protocol-techno>

2.1 Terminology and Class Definitions

The definition in Figure ?? introduces classes for vehicles, days and roads.

```

class Vehicle {
  weight: Integer
}
class Car extends Vehicle {
  doors: Integer
}
class Truck extends Vehicle
class SportsCar extends Car

class Day
class Workday extends Day
class Holiday extends Day

class Road
class Highway extends Road

```

Fig. 1: Class definitions of speedlimit example

Classes are arranged in a tree-shaped hierarchy, having a class named `Class` as its top element. Classes that are not explicitly derived from another class via `extends` are implicitly derived from `Class`. A class S derived from a class C by `extends` will be called a subclass of C , and the immediate subclasses of `Class` will be called *sorts* in the following. Intuitively, classes are meant to be sets of entities, with subclasses being interpreted as subsets. Different subclasses of a class are not meant to be disjoint.

Class definitions can come with attributes, in braces. These attributes can be of simple type, as in the given example, or of higher type (the notion of type will be explained in Section ??). In a declarative reading, attributes can be understood as a shorthand for function declarations that have the class they are defined in as additional domain. Thus, the attribute `weight` corresponds to a top-level declaration `weight: Vehicle -> Integer`. In a more operational reading, L4 classes can be understood as prototypes of classes in object-oriented programming languages, and an alternative field selection syntax can be used: For $v: \text{Vehicle}$, the expression `v.weight` is equivalent to `weight(v)`, at least logically, even though the operational interpretations may differ.

2.2 Types and Function Declarations

L4 is an *explicitly* and *strongly typed* language: all entities such as functions, predicates and variables have to be declared before being used. One purpose of this measure is to ensure that the executable sublanguage of L4, based on the simply-typed lambda calculus with subtyping, enjoys a type soundness property: evaluation of a function cannot produce a dynamic type error.

Figure ?? shows two function declarations. Functions with `Boolean` result type will sometimes be called *predicates* in the following, even though there is no syntactic difference. All the declared classes are considered as elementary types, as well as `Integer`, `Float`, `String` and `Boolean` (which are internally also treated as classes). If T_1, T_2, \dots, T_n are types, then so are function types $T_1 \rightarrow T_2$ and tuple types (T_1, \dots, T_n) . The type system and the expression language, to be presented later, are higher-order, but extraction to some solvers (Section ??) will be limited to (restricted) first-order theories.

```
decl isCar : Vehicle -> Boolean
decl maxSp : Vehicle -> Day -> Road -> Integer -> Boolean
```

Fig. 2: Declarations of speedlimit example

The nexus between the terminological and the logical level is established with the aid of *characteristic predicates*. Each class C which is a subclass of sort S gives rise to a declaration `isC`: $S \rightarrow \text{Boolean}$. An example is the declaration of `isCar` in Figure ?. In the L4 system, this declaration, as well as the corresponding class inclusion axiom (Section ??), are generated automatically.

Two classes derived from the same base class (thus: C_1 extends B and C_2 extends B) are not necessarily disjoint.

From the subclass relation, a *subtype* relation \preceq can be defined inductively as follows: if C extends B , then $C \preceq B$, and for types $T_1, \dots, T_n, T'_1, \dots, T'_n$, if $T_1 \preceq T'_1, \dots, T_n \preceq T'_n$, then $T_1 \rightarrow T_2 \preceq T'_1 \rightarrow T'_2$ and $(T_1, \dots, T_n) \preceq (T'_1, \dots, T'_n)$.

Without going into details of the type system, let us remark that it has been designed to be compatible with subtyping: if an element of a type is acceptable in a given context, then so is an element of a subtype. In particular,

- for field selection, if C' is a class having field f of type T , and $C \preceq C'$, and $c : C$, then field selection is well-typed with $c.f : T$.
- for function application, if $f : A' \rightarrow B$ and $a : A$ and $A \preceq A'$, then function application is well-typed with $f a : B$.

2.3 Rules

Before discussing rules, a few remarks about *expressions* which are their main constituents: L4 supports a simple functional language featuring typical arithmetic, Boolean and comparison operators, an `if .. then .. else` expression, function application, anonymous functions (*i.e.*, lambda abstraction) written in the form `\x : T -> e`, class instances and field access (as mentioned before). A *formula* is just a Boolean expression, and, consequently, so are quantified formulas `forall x:T. form` and `exists x:T. form`.

In its most complete form, a *rule* is composed by a list of variable declarations introduced by the keyword `for`, a precondition introduced by `if` and a postcondition introduced by `then`. Figure ?? gives an example of rules of our

MS: Promise

speed limit scenario, stating, respectively, that the maximal speed of cars is 90 km/h on a workday, and that they may drive at 130 km/h if the road is a highway. Note that in general, both pre- and postconditions are Boolean formulas that can be arbitrarily complex, thus are not limited to conjunctions of literals in the preconditions or atomic formulas in the postconditions.

MS: scenario
changed wrt repo

```

rule <maxSpCarWorkday>
  for v: Vehicle, d: Day, r: Road
    if isCar v && isWorkday d
      then maxSp v d r 90

rule <maxSpCarHighway>
  for v: Vehicle, d: Day, r: Road
    if isCar v && isHighway r
      then maxSp v d r 130

```

Fig. 3: Rules of speedlimit example

Rules whose precondition is true can be written as

```
fact <name> for v1: T1 ... vn: Tn P v1...vn
```

Rules may not contain free variables, so all variables occurring in the body of the rule have to be declared in the for clause. In the absence of variables to be declared, the for clause can be omitted.

Intuitively, a rule

```
rule <r> for  $\vec{v}: \vec{T}$  if Pre  $\vec{v}$  then Post  $\vec{v}$ 
```

corresponds to a universally quantified formula $\forall \vec{v}: \vec{T}. Pre \vec{v} \rightarrow Post \vec{v}$ that could directly be written as a fact, and it may seem that a separate rule syntax is redundant. This is not so, because the specific structure of rules makes them amenable to transformations that are useful for defeasible reasoning, as will be seen in Section ??.

Apart from user-defined rules and rules obtained by transformation, there are system generated rules: For each subclass relation C extends B , a class inclusion axiom of the form for x: S if is C x then is B x is generated, where is C and is B are the characteristic predicates and S is the common supersort of C and B .

MS: Promise

2.4 Assertions

Assertions are statements that the L4 system is meant to verify or to reject – differently said, they are proof obligations. These assertions are verified relative to a rule set comprising some or all of the rules and facts stated before.

The statement in Figure ?? claims that the predicate maxSp behaves like a function, *i.e.* given the same car, day and road, the speed will be the same.

Instead of a universal quantification, we here use variables `inst...` that have been declared globally, because they produce more readable (counter-)models.

```
assert <maxSpFunctional> {SMT: {valid}}
  maxSp instCar instDay instRoad instSpeed1 &&
  maxSp instCar instDay instRoad instSpeed2
  --> instSpeed1 == instSpeed2
```

Fig. 4: Assertions of speedlimit example

The active rule set used for verification can be configured, by adding rules to or deleting rules from a default set. Assume the active rule set consists of n rules whose logical representation is $R_1 \dots R_n$, and assume the formula of the assertion is A . The proof obligation can then be checked for

- *satisfiability*: in this case, $R_1 \wedge \dots \wedge R_n \wedge A$ is checked for satisfiability.
- *validity*: in this case, $R_1 \wedge \dots \wedge R_n \longrightarrow A$ is checked for validity.

In either case, if the proof fails, a model resp. countermodel is produced. In the given example, the SMT solver checks the validity of the formula and indeed returns a countermodel that leads to contradictory prescriptions of the maximal speed: if the vehicle is a car, the day a workday and the road a highway, the maximal speed can be 90 or 130, depending on the rule applied.

MS: Put output of model checker here?

The assertion `maxSpFunctional` can be considered as an essential consistency requirement and a rule system violating it as inconsistent. One remedial action is to declare one of the rules as default and the other rule as overriding it. In Section ??, we will discuss different solutions implemented in L4.

After this repair action, `maxSpFunctional` will be provable (under additional natural conditions described in Section ??). We can now continue to probe other consistency requirements, such as exhaustiveness stating that a maximal speed is defined for every combination of vehicle:

```
assert <maxSpExhaustive>
  exists sp: Integer. maxSp instVeh instDay instRoad sp
```

The intended usage scenario of L4 is that by an interplay of proving assertions and repairing broken rules, one arrives at a rule set satisfying general principles of coherence, completeness and other, more elusive properties such as fairness (at most temporary exclusion from essential resources or rights).

MS: maybe put this into the intro

3 Reasoning with and about Rules

3.1 Facets of Defeasible Reasoning

To be done

Discussion on existing notions of defeasibility in particular in a legal context

- [?]
- [?]
- [?]
- More recent: [?]
- Using SMT solvers for CASP: [?]. Some of the techniques are similar (use of Clark completion), but our purpose is not to simulate answer set programming in SMT solvers, but to represent legal reasoning. In this context, maybe also look at [?].

Given a plethora of different notions of “defeasibility”, we had to make a choice as to which notions to support, and which semantics to give to them. We will here concentrate on two concepts, which we call *rule modifiers*, that limit the applicability of rules and make them “defeasible”. They will be presented in Section ??, and we will see how to give them a semantics in a classical first-order logic, but also in a non-monotonic logic in a system based on Answer Set Programming (Section ??). An orthogonal question is that of arriving at conclusion in absence of complete information, which, via the mechanism of negation as failure, often prones the use of non-monotonic logics. In Section ??, we will see how similar effects can be achieved by means of *rule inversion*. An Answer Set Programming approach to defeasible reasoning, with an emphasis on rule modifiers, will be presented in Section ??. We finish with a comparison of the two strands of reasoning in Section ??.

MS: as/by??

3.2 Introducing Rule Modifiers

We will concentrate on two rule modifiers that restrict the applicability of rules and that frequently occur in law texts: *subject to* and *despite*. To illustrate their use, we consider an excerpt of Singapore’s Professional Conduct Rules § 34 [?] (also see [?] for a more detailed treatment of this case study):

- (1) A legal practitioner must not accept any executive appointment associated with any of the following businesses:
 - (a) any business which detracts from, is incompatible with, or derogates from the dignity of, the legal profession;
 - (b) any business which materially interferes with the legal practitioner’s primary occupation of practising as a lawyer; (...)

- (5) Despite paragraph (1)(b), but subject to paragraph (1)(a) and (c) to (f), a locum solicitor may accept an executive appointment in a business entity which does not provide any legal services or law-related services, if all of the conditions set out in the Second Schedule are satisfied.

The two main notions developed in the Conduct Rules are which executive appointments a legal practitioner *may* or *must not* accept under which circumstances. As there is currently no direct support for deontic logics in L4, these notions are defined as two predicates `MayAccept` and `MustNotAccept`, with the intended meaning that these two notions are contradictory, and this is indeed what will be provable after a complete formalization.

Let us here concentrate on the modifiers *despite* and *subject to*. A synonym of “despite” that is often used in legal texts is “notwithstanding”, and a synonym of “subject to” is “except as provided in”, see [?].

The reading of rule (5) is the following:

- “subject to paragraph (1)(a) and (c) to (f)” means: rule (5) applies as far as (1)(a) and (c) to (f) is not established. Differently said, rules (1)(a) and (c) to (f) undercut or defeat rule (5).

One way of explicating the “subject to” clause would be to rewrite (5) to: “Despite paragraph (1)(b), provided the business does not detract from, is incompatible with, or derogate from the dignity of, the legal profession; and provided that not [clauses (1)(c) to (f)]; then a locum solicitor⁶ may accept an executive appointment.”

- “despite paragraph (1)(b)” expresses that rule (5) overrides rule (1)(b). In a similar spirit as the “subject to” clause, this can be made explicit by introducing a proviso, however not locally in rule (5), but remotely in rule (1)(b).

One way of explicating the “despite” clause of rule (5) would be to rewrite (1)(b) to: “Provided (5) is not applicable, a legal practitioner must not accept any executive appointment associated with any business which materially interferes with the legal practitioner’s primary occupation of practising as a lawyer.”

The astute reader will have remarked that the treatment in both cases is slightly different, and this is not related to the particular semantics of *subject to* and *despite*: we can state defeasibility

- either in the form of (negated) preconditions of rules: “rule r_1 is applicable if the preconditions of r_2 do not hold”;
- or in the form of (negated) derivability of the postcondition of rules: “rule r_1 is applicable if the postcondition of r_2 does not hold”.

MS: where?

We will subsequently come back to this difference.

Before looking at a formalization, let us summarize this informal exposition of defeasibility rule modifiers as follows:

MS: Make the writing of the modifiers more homogenous: in italics or in quotes

⁶ in our class-based terminology, a subclass of legal practitioner

- “ r_1 subject to r_2 ” and “ r_1 despite r_2 ” are complementary ways of expressing that one rule may override the other rule. They have in common that r_1 and r_2 have contradicting conclusions. The conjunction of the conclusions can either be directly unsatisfiable (may accept vs. must not accept) or unsatisfiable *w.r.t.* an intended background theory (obtaining different maximal speeds is inconsistent when expecting maxSp to be functional in its fourth argument).
- Both modifiers differ in that “subject to” modifies the rule to which it is attached, whereas “despite” has a remote effect.
- They permit to structure a legal text, favoring conciseness and modularity: In the case of *despite*, the overridden, typically more general rule need not be aware of the overriding, typically subordinate rules.
- Even though these modifiers appear to be mechanisms on the meta-level in that they reasoning about rules, they can directly be reflected on the object-level.

Making the meaning of the modifiers explicit can therefore be understood as a *compilation* problem, which will be described in the following.

In L4, rule modifiers are introduced with the aid of *rule annotations*, with a list of rule names following the keywords `subjectTo` and `despite`. We return to our running example and modify rule `maxSpCarHighway` of Figure ?? with

```
rule <maxSpCarHighway>
  {restrict: {despite: maxSpCarWorkday}}
# rest of rule unchanged
```

For the delight of the public of the country with the highest density of sports cars, we also introduce a new rule:

```
rule <maxSpSportsCar>
  {restrict: {subjectTo: maxSpCarWorkday,
             despite: maxSpCarHighway}}
  for v: Vehicle, d: Day, r: Road
  if isSportsCar v && isHighway r
  then maxSp v d r 320
```

MS: Problem with spaces in `lstlisting`

In the following, we will examine the interaction of these rules.

4 Defeasible Reasoning in a Classical Logic

4.1 Rule Modifiers in Classical Logic

In this section, we will describe how to rewrite rules, progressively eliminating the instructions appearing in the rule annotations so that in the end, only purely logical rules remain. Whereas the first preprocessing steps (Section ??) are generic, we will discuss two variants of conversion into logical format (Sections ?? and ??).

4.1.1 Preprocessing Preprocessing consists of several elimination steps that are carried out in a fixed order.

“Despite” elimination As can be concluded from the discussion in Section ??, a **despite** r_2 clause appearing in rule r_1 is equivalent to a **subjectTo** r_1 clause in rule r_2 . The first rule transformation consists in applying exhaustively the following *despite elimination* rule transformer:

MS: In the whole discussion (and the implementation), make a clearer distinction between rule set transformer **restrict** and rule generator / transformer **derived**.

despiteElim:

$$\{r_1\{\text{restrict} : \{\text{despite } r_2\} \uplus a_1\}, r_2\{\text{restrict} : a_2\}, \dots\} \longrightarrow \{r_1\{\text{restrict} : a_1\}, r_2\{\text{restrict} : \{\text{subjectTo } r_1\} \uplus a_2\}, \dots\}$$

Example 1.

Application of this rewrite rule to the three example rules `maxSpCarWorkday`, `maxSpCarHighway` and `maxSpSportsCar` changes them to:

```
rule <maxSpCarWorkday>
  {restrict: {subjectTo: maxSpCarHighway}}
rule <maxSpCarHighway>
  {restrict: {subjectTo: maxSpSportsCar}}
rule <maxSpSportsCar>
  {restrict: {subjectTo: maxSpCarWorkday}}
```

Here, only the headings are shown, the bodies of the rules are unchanged.

One defect of the rule set already becomes apparent to the human reader at this point: the circular dependency of the rules. We will however continue with our algorithm, applying the next step which will be to rewrite the `{restrict: {subjectTo: ...}}` clauses. Please note that each rule can be **subjectTo** several other rules, each of which may have a complex structure as a result of transformations that are applied to it.

“Subject” To elimination The rule transformer *subjectToElim* does the following: it splits up the rule into two rules, (1) its source (the rule body as originally given), and (2) its definition as the result of applying a rule transformation function to several rules.

Example 2. Before stating the rule transformer, we show its effect on rule `maxSpCarWorkday` of Example ??. On rewriting with *subjectToElim*, the rule is transformed into two rules:

```
# new rule name, body of rule unchanged
rule <maxSpCarWorkday'Orig>
  {source}
  for v: Vehicle, d: Day, r: Road
```

```

if isCar v && isWorkday d
then maxSp v d r 90

# rule with header and without body
rule <maxSpCarWorkday>
{derived: {apply:
{restrictSubjectTo maxSpCarWorkday'Orig maxSpSportsCar}}}}

```

MS: check syntax of apply

We can now state the transformation (after grouping the `subjectTo` $r_2, \dots, \text{subjectTo } r_n$ into `subjectTo` $[r_2 \dots r_n]$):

subjectToElim:

$$\{r_1\{\text{restrict} : \{\text{subjectTo } [r_2, \dots, r_n]\}\}, \dots\} \longrightarrow \{r_1^o\{\text{source}\}, r_1\{\text{derived} : \{\text{apply} : \{\text{restrictSubjectTo } r_1^o [r_2 \dots r_n]\}\}\}, \dots\}$$

Computation of derived rule The last step consists in generating the derived rules, by evaluating the value of the rule transformer expression marked by `apply`. The rules appearing in these expressions may themselves be defined by complex expressions. However, direct or indirect recursion is not allowed. For simplifying the expressions in a rule set, we compute a rule dependency order \preceq_R defined by: $r \preceq_R r'$ iff r appears in the defining expression of r' . If \preceq_R is not a partial order (in particular, if it is not cycle-free), then evaluation fails. Otherwise, we order the rules topologically by \preceq_R and evaluate the expressions starting from the minimal elements.

Example 3. It is at this point that the cyclic dependence already remarked after Example ?? will be discovered. We have:

```

maxSpCarWorkday'Orig, maxSpCarHighway  $\preceq_R$  maxSpCarWorkday
maxSpCarHighway'Orig, maxSpSportsCar  $\preceq_R$  maxSpCarHighway
maxSpSportsCar'Orig, maxSpCarWorkday  $\preceq_R$  maxSpSportsCar

```

which cannot be totally ordered.

Let us fix the problem by changing the heading of rule `maxSpCarHighway` from `despite` to `subjectTo`:

```

rule <maxSpCarHighway>
{restrict: {subjectTo: maxSpCarWorkday}}

```

After rerunning *despiteElim* and *subjectToElim*, we can now order the rules: $\{\text{maxSpSportsCar'Orig maxSpCarHighway'Orig, maxSpCarWorkday}\} \preceq_R \text{maxSpSportsCar} \preceq_R \text{maxSpCarHighway}$ and will use this order for rule elaboration.

4.1.2 Restriction via Preconditions Here, we propose one possible implementation of the rule transformer `restrictSubjectTo` introduced in Section ?? that takes a rule r_1 and a list of rules $[r_2 \dots r_n]$ and produces a new rule, by adding the negation of the preconditions of $[r_2 \dots r_n]$ to r_1 . More formally:

- `restrictSubjectTo` $r_1 \sqcup = r_1$
- `restrictSubjectTo` $r_1 (r' \uplus rs) =$
`restrictSubjectTo` $(r_1(\text{precond} := \text{precond}(r_1) \wedge \neg \text{precond}(r')) rs$

where $\text{precond}(r)$ selects the precondition of rule r and $r(\text{precond} := p)$ updates the precondition of rule r with p .

There is one proviso to the application of `restrictSubjectTo`: the rules have to have the same *parameter interface*: the number and types of the parameters in the rules' `for` clause have to be the same. Rules with different parameter interfaces can be adapted via the `remap` rule transformer. The rule

MS: Promise

```
rule <r> for  $x_1:T_1 \dots x_n:T_n$  if  $\text{Pre}(x_1, \dots, x_n)$  then  $\text{Post}(x_1, \dots, x_n)$ 
```

is remapped with

```
remap r [ $y_1:S_1, \dots, y_m:S_m$ ] [ $x_1 := e_1, \dots, x_n := e_n$ ]
```

to the rule

```
rule <r> for  $y_1:S_1 \dots y_m:S_m$  if  $\text{Pre}(e_1, \dots, e_n)$  then  $\text{Post}(e_1, \dots, e_n)$ 
```

Here, e_1, \dots, e_n are expressions that have to be well-typed with types E_1, \dots, E_n in context $y_1:S_1, \dots, y_m:S_m$ (which means in particular that they may contain the variables y_i) with $E_i \preceq T_i$ (with the consequence that the pre- and post-conditions of the new rule remain well-typed).

Example 4.

We come back to the running example. When processing the rules in the order of \preceq_R , rule `maxSpSportsCar`, defined by `apply: {restrictSubjectTo maxSpSportsCar'Orig maxSpCarWorkday}`, becomes:

```
rule <maxSpSportsCar>
  for v: Vehicle, d: Day, r: Road
  if isSportsCar v && isHighway r &&
    not (isCar v && isWorkday d)
  then maxSp v d r 320
```

We can now state `maxSpCarHighway`, which has been defined by `apply: {restrictSubjectTo maxSpCarHighway'Orig maxSpSportsCar}`, as:

```
rule <maxSpCarHighway>
  for v: Vehicle, d: Day, r: Road
  if isCar v && isHighway r &&
    not (isSportsCar v && isHighway r &&
      not (isCar v && isWorkday d)) &&
    not (isCar v && isWorkday d)
  then maxSp v d r 130
```

One downside of the approach of adding negated preconditions is that the preconditions of rules can become very complex. This effect is mitigated by the

fact that conditions in `subjectTo` and `despite` clauses express specialization or refinement and often permit substantial simplifications. Thus, the precondition of `maxSpSportsCar` simplifies to `isSportsCar v && isHighway r && isWorkday d` and the precondition of `maxSpCarHighway` to `isCar v && isHighway r && not (isSportsCar v && isWorkday d)`.

4.1.3 Restriction via Derivability We now give an alternative reading of `restrictSubjectTo`. To illustrate the point, let us take a look at a simple propositional example.

Example 5. Take the definitions:

```
rule <r1> if B1 then C1
rule <r2> {subjectTo: r1} if B2 then C2
```

Instead of saying: `r2` corresponds to `if B2 && not B1 then C2` as in Section ??, we would now read it as “if the conclusion of `r1` cannot be derived”, which could be written as `if B2 && not C1 then C2`. The two main problems with this naive approach are the following:

- As mentioned in Section ??, a *subject to* restriction is often applied to rules with contradicting conclusions, so in the case that `C1` is not `C2`, the generated rule would be a tautology.
- In case of the presence of a third rule

```
rule <r3> if B3 then C1
```

a derivation of `C1` from `B3` would also block the application of `r2`, and `subjectTo: r1` and `subjectTo: r1, r3` would be indistinguishable.

We now sketch a solution for rule sets whose conclusion is always an atom (and not a more complex formula).

1. In a preprocessing stage, all rules are transformed as follows:
 - (a) We assume the existence of a class `Rulename`, which we will take to be `String` in the following.
 - (b) All the predicates p occurring in the conclusions of rules (called *transformable predicates*) are converted into predicates p^+ with one additional argument of type `Rulename`. In the example, $C1^+$: `Rulename -> Boolean` and similarly for `C2`.
 - (c) The transformable predicates p in conclusions of rules receive one more argument, which is the name rn of the rule: p is transformed into $p^+ rn$. The informal reading is “the predicate is derivable with rule rn ”.
 - (d) All transformable predicates in the preconditions of the rules receive one more argument, which is a universally quantified variable of type `Rulename` bound in the `for`-list of the rule.

2. In the main processing stage, `restrictSubjectTo` in the rule annotations generates rules according to:
- `restrictSubjectTo r_1 [] = r_1`
 - `restrictSubjectTo r_1 ($r' \uplus rs$) =`
`$\text{restrictSubjectTo } (r_1(\text{precond} := \text{precond}(r_1) \wedge \neg \text{postcond}(r')))$ rs`
 Thus, the essential difference *w.r.t.* the definition of Section ?? is that we add the negated postcondition.

Example 6. The rules of Example ?? are now transformed to:

```
rule <r1> for rn: Rulename if B1+ rn then C1+ r1
rule <r2> for rn: Rulename if B2+ rn and not C1+ r1 then C2+ r2
```

The derivability of another instance of $C1$, such as $C1^+ r3$, would not inhibit the application of $r2$ any more.

Example 7. The two rules of the running example become, after resolution of the `restrictSubjectTo` clauses:

```
rule <maxSpSportsCar>
  for v: Vehicle, d: Day, r: Road
  if isSportsCar v && isHighway r &&
    not maxSp+ maxSpCarWorkday v d r 90
  then maxSp v d r 320
rule <maxSpCarHighway>
  for v: Vehicle, d: Day, r: Road
  if isCar v && isHighway r &&
    not maxSp+ maxSpCarWorkday v d r 90 &&
    not maxSp+ maxSpSportsCar v d r 320
  then maxSp v d r 130
```

4.2 Rule Inversion

The purpose of this section is to derive formulas that, for a given rule set, simulate negation as failure, but are coded in a classical first-order logic, do not require a dedicated proof engine (such as Prolog) and can be checked with a SAT or SMT solver. The net effect is similar to the completion introduced by Clark [?]; however, the justification is not operational as in [?], but takes inductive closure as a point of departure. Apart from that, our technique applies to a considerably wider class of formulas.

In the following, we assume that our rules have an atomic predicate P as conclusion, whereas the precondition Pre can be an arbitrarily complex formula. We furthermore assume that rules are in *normalized form*: P may only be applied to n distinct variables x_1, \dots, x_n , where n is the arity of P , and the rule quantifies over exactly these variables. For notational simplicity, we write normalized rules in logical format, ignoring types: $\forall x_1, \dots, x_n. Pre(x_1, \dots, x_n) \longrightarrow Post(x_1, \dots, x_n)$.

Every rule can be written in normalized form, by applying the following algorithm:

- Remove expressions or duplicate variables in the conclusion, by using the equivalences $P(\dots e \dots) = (\forall x.x = e \longrightarrow P(\dots x \dots))$ for a fresh variable x , and similarly $P(\dots y \dots y \dots) = (\forall x.x = y \longrightarrow P(\dots x \dots y \dots))$.
- Remove variables from the universal quantifier prefix if they do not occur in the conclusion, by using the equivalence $(\forall x.Pre(\dots x \dots) \longrightarrow P) = (\exists x.Pre(\dots x \dots) \longrightarrow P)$.

For any rule set R and any predicate P , we can form the set of P -rules $\{\forall x_1, \dots, x_n. Pre_1(x_1, \dots, x_n) \longrightarrow P(x_1, \dots, x_n), \dots, \forall x_1, \dots, x_n. Pre_k(x_1, \dots, x_n) \longrightarrow P(x_1, \dots, x_n)\}$ as the subset of R containing all rules having P as post-condition.

The inductive closure of a set of P -rules is the predicate P^* defined by the second-order formula

$$P^*(x_1, \dots, x_n) = \forall P. (Pre_1(x_1, \dots, x_n) \longrightarrow P(x_1, \dots, x_n)) \longrightarrow \dots \\ (Pre_k(x_1, \dots, x_n) \longrightarrow P(x_1, \dots, x_n)) \longrightarrow \\ P(x_1, \dots, x_n)$$

P^* can be understood as the least predicate satisfying the set of P -rules and is the predicate that represents “all that is known about P and assuming nothing else about P is true”, and corresponds to the notion of exhaustiveness prevalent in law texts. It can also be understood as the static equivalent of the operational concept of negation as failure for predicate P .

As an illustration, let us note that in case the P -rule set is empty, *i.e.* there are no rules establishing P , we have $P^*(x_1, \dots, x_n) = \forall P. P(x_1, \dots, x_n) = \perp$.

Obviously, a second-order formula such as the definition of P^* is unwieldy in fully automated theorem proving. We derive one consequence:

Lemma 1. $P^*(x_1, \dots, x_n) \longrightarrow Pre_1(x_1, \dots, x_n) \vee \dots \vee Pre_k(x_1, \dots, x_n)$

Proof. Expand the definition of P^* and instantiate the universal variable P with $\lambda x_1 \dots x_n. Pre_1(x_1, \dots, x_n) \vee \dots \vee Pre_k(x_1, \dots, x_n)$.

As a consequence of the Löwenheim–Skolem theorem, there is no first-order equivalent of P^* : a formula of the form P^* can characterize the natural numbers up to isomorphism, but no first-order formula can. In the absence of such a first-order equivalent, we define the formula

$$\forall x_1, \dots, x_n. P(x_1, \dots, x_n) \longrightarrow Pre_1(x_1, \dots, x_n) \vee \dots \vee Pre_k(x_1, \dots, x_n)$$

called the *inversion formula of P* , as an approximation of the effect of P^* .

Inversion formulas can be automatically derived and added to the rule set in L4 proofs; they turn out to be essential for consistency properties. For example, the functionality of `maxSp` stated in Figure ?? is not provable without the inversion formula of `maxSp`.

To avoid misunderstandings, we should emphasize that this approach is entirely based on a classical monotonic logic, in spite of non-monotonic effects. Adding a new P -rule may invalidate previously provable facts, but this is only so because the new rule alters the inversion formula of P .

4.3 Comparison

One may wonder whether, starting from the same set of rules, the transformations in Section ?? and Section ?? produce equivalent rules. On the face of it, this is not so, because the transformation via derivability modifies the arity of the predicates, so the rule sets have different models.

We will however show that the two rule sets have corresponding sets of models. This will be made more precise in the following. To fix notation, assume \mathcal{R}_M to be a set of rules annotated with rule modifiers. Let \mathcal{R}_P be the set of rules obtained from \mathcal{R}_M through the rule translation via preconditions of Section ??, and similarly \mathcal{R}_D the set of rules obtained from \mathcal{R}_M through the rule translation via derivability of Section ?. From these rule sets, we obtain formula sets \mathcal{F}_P respectively \mathcal{F}_D by

- translating rules to formulas;
- adding inversion formulas Inv_C for all the transformable predicates C of the rule set;

MS: introduce that notation

Lemma 2. *Any model \mathcal{M}_P of \mathcal{F}_P can be transformed into a model \mathcal{M}_D of \mathcal{F}_D .*

Proof. We consider the transformation of a model \mathcal{M}_P to a model \mathcal{M}_D , and assume \mathcal{M}_P is a model of \mathcal{F}_P . We now construct an interpretation \mathcal{M}_D for the formulas with the signature over \mathcal{F}_D .

The interpretation \mathcal{M}_D will be the same as \mathcal{M}_P , except for (1) the interpretation of the new types RuleName_C , each of which will be chosen to be the set of all rule names having C as conclusion, and (2) the interpretation of the new predicates C^+ on which we will focus now: For each rule $\forall x_1, \dots, x_n. \text{Pre}(x_1, \dots, x_n) \rightarrow C(x_1, \dots, x_n)$ with name rn , whenever the n -tuple (a_1, \dots, a_n) satisfies the precondition Pre under \mathcal{M}_P and, consequently, $(a_1, \dots, a_n) \in C^{\mathcal{M}_P}$, we will have $(rn, a_1, \dots, a_n) \in (C^+)^{\mathcal{M}_D}$.

It remains to be shown that \mathcal{M}_D is indeed a model of \mathcal{M}_D . We show that related formulas in \mathcal{F}_P and \mathcal{F}_D are interpreted as true in \mathcal{M}_P resp. \mathcal{M}_D , where two formulas are *related* if they are rules originating from the same rule of \mathcal{R}_M , or if they are related inversion predicates Inv_C and Inv_{C^+} .

We first address related rules. The proof is by well-founded induction over the rule order \preceq_R . Consider a rule $r_P \in \mathcal{F}_P$ with rule name rn_P which by construction has the form $r_P = \forall x_1, \dots, x_n. \text{pre}_P^o \wedge \neg \text{pre}_P^1 \wedge \neg \text{pre}_P^k \rightarrow C(x_1, \dots, x_n)$. We make a case distinction:

- Assume that for arguments (a_1, \dots, a_n) , interpretation \mathcal{M}_P satisfies the precondition $\text{pre}_P^o \wedge \neg \text{pre}_P^1 \wedge \neg \text{pre}_P^k$ and thus also the conclusion. In this case, $(rn_P, a_1, \dots, a_n) \in (C^+)^{\mathcal{M}_D}$, thus satisfying the related rule $r_D \in \mathcal{F}_D$.

- Assume that for arguments (a_1, \dots, a_n) , interpretation \mathcal{M}_P does not satisfy the precondition. Either pre_P^o is not satisfied, leading again to a satisfying assignment of the related rule r_D , or one of the pre_P^i is satisfied. In this case, as the rule r_P^i with precondition pre_P^i is strictly smaller than r_P w.r.t. \preceq_R , by induction hypothesis, also the postcondition of r_P^i will be satisfied, so that in \mathcal{M}_D , one negated precondition of the related rule r_D is not satisfied, so r_D is satisfied.

Once the equi-satisfiability of related rules has been established, it is easy to do so for related inversion predicates Inv_C and Inv_{C+} .

Lemma 3. *Any model \mathcal{M}_D of \mathcal{F}_D can be transformed into a model \mathcal{M}_P of \mathcal{F}_P .*

Proof. (Sketch) In analogy to Lemma ??, we start from a model \mathcal{M}_D of \mathcal{F}_D and construct a model \mathcal{M}_P of \mathcal{F}_P .

As in Lemma ??, the proof is by induction on \preceq_R . Consider a rule $r_D \in \mathcal{F}_D$ with rule name rn_D which by construction has the form $r_D = \forall x_1, \dots, x_n. pre_D^o \wedge \neg post_D^1(rn_1) \wedge \neg post_D^k(rn_k) \longrightarrow C^+(rn_D, x_1, \dots, x_n)$. Again, we make a case distinction:

- Assume that for arguments (a_1, \dots, a_n) , interpretation \mathcal{M}_D satisfies the precondition and thus also the conclusion. In this case, $(a_1, \dots, a_n) \in C^{\mathcal{M}_P}$, thus satisfying the related rule $r_P \in \mathcal{F}_P$.
- Assume that for arguments (a_1, \dots, a_n) , interpretation \mathcal{M}_D does not satisfy the precondition. The interesting situation is if one $post_D^i(rn_i)$ is satisfied. At this point, we need the inversion formula of $post_D^i$, of the form $\forall r. post_D^i(r) \longrightarrow P_1(r) \vee \dots \vee P_p(r)$. The rule name rn_i permits to select precisely the precondition P_j of the related formula $r_P = \forall x_1, \dots, x_n. pre_P^o \wedge \neg pre_P^1 \wedge \neg pre_P^k \longrightarrow C(x_1, \dots, x_n)$.

We should emphasize that, in the proof of Lemma ??, the inversion formulas play a decisive role.

5 Defeasible Reasoning with Answer Set Programming

5.1 Introduction

The purpose of this section is to give an account of the work we have been doing using Answer Set Programming to formalize and reason about legal rules. This approach is complementary to the one described before using SMT solvers. Here we will not go too much into the details of how various L4, language constructs map to the ASP formalisation. Our intention rather, is to present how some core legal reasoning tasks can be implemented in ASP while keeping the ASP representation readable, intuitive and respecting the idea of having an ‘isomorphism’ between the rules and the encoding. Going forward, our intention is to develop a method to compile L4 code to a suitable ASP representation like the one we shall now present. We formalize the notion of what it means to ‘satisfy’ a rule

set. We will do this in a way that is most amenable to ASP.

First we shall give a brief overview of Answer Set Programming (ASP). ASP is a declarative programming language used mainly in Knowledge Representation and Reasoning to model rules, facts, integrity constraints etc. within a particular scenario that one wishes to consider. A rule in ASP has the form:

$$h \leftarrow b_1, b_2, \dots, b_k, \text{not } b_{k+1}, \dots, \text{not } b_n.$$

Here h and b_1, \dots, b_n are atoms. For an atom b_i , $\text{not } b_i$ is the negated atom where the *not* represents negation as failure. Informally $\text{not } b_i$ is true exactly when b_i cannot be derived. This is also sometimes known as the ‘closed world assumption’. Intuitively the rule above says that when $b_1, b_2, \dots, b_k, \text{not } b_{k+1}, \dots, \text{not } b_n$ are all true, h is true. h is also sometimes known as the head of the rule and the positive and negated atoms $b_1, b_2, \dots, b_k, \text{not } b_{k+1}, \dots, \text{not } b_n$ form the body. A rule with only a head and an empty body is called a fact. A logic program is a set of facts and rules. (In fact ASP can also model other things like integrity constraints, disjunctions in rule heads etc, but we will not be using these features in our paper). When a logic program is passed to an ASP solver, the solver returns a set of *stable models* (also known as *answer sets*) which make all the rules and facts in the logic program true. The set of *stable models* of a logic program is calculated using the *stable model semantics* for ASP. For logic programs without negation-as-failure, the set of stable models is exactly the set of subset minimal models of the program. For logic programs with negation as failure stable models are most commonly defined using a construction known as the *reduct* of a logic program with respect to an *herbrand interpretation*. Please see [?] for more details on ASP and the stable model semantics.

5.2 Formal Setup

Let the tuple $Config = (R, F, M, I)$ denote a *configuration* of legal rules. The set R denotes a set of rules of the form $pre_con(r) \rightarrow concl(r)$. These are ‘naive’ rules with no information pertaining to any of the other rules in R . F is a set of positive atoms and predicates that describe facts of the legal scenario we wish to consider. M is a set of the binary predicates *despite*, *subject_to* and *strong_subject_to*. I is a collection of minimal inconsistent sets of positive atoms/predicates. Henceforth for a rule r , we may write C_r for its conclusion $Concl(r)$.

Note that, throughout this section, given any rule r C_r is assumed to be a single positive atom. That is there are no disjunctions or conjunctions in rule conclusions. Also any rule pre-condition ($pre_con(r)$) is assumed to be a conjunction of positive and negated atoms. Here negation denotes ‘negation as failure’.

Throughout this document, whenever we use an uppercase or lowercase letter (like r , r_1 , R etc.) to denote a rule that is an argument, in a binary predicate, we mean the unique numeric or alpha-numeric rule id associated with that rule. The binary predicate *legally_valid*(r, c) intuitively means that the rule r is

'in force' and it has conclusion c . The unary predicate $is_legal(c)$ intuitively means that c legally holds/has legal status. The predicates $despite$, $subject_to$ and $strong_subject_to$ all cause some rules to override others. Their precise properties will be given next.

5.3 Semantics

A set S of is_legal and $legally_valid$ predicates is called a *legal model* of $Config = (R, F, M, I)$, if and only if

- (A1) $\forall f \in F \ is_legal(f) \in S$.
- (A2) $\forall r \in R$, if $legally_valid(r, C_r) \in S$, then $S \models is_legal(pre_con(r))$ and $S \models is_legal(C_r)$ ⁷
- (A3) $\forall c$, if $is_legal(c) \in S$, then either $c \in F$ or there exists $r \in R$ such that $legally_valid(r, C_r) \in S$ and $c = C_r$.
- (A4) $\forall r_1, r_2 \in R$, if $despite(r_1, r_2) \in M$ and $S \models is_legal(pre_con(r_2))$, then $legally_valid(r_1, C_{r_1}) \notin S$
- (A5) $\forall r_1, r_2 \in R$, if $strong_subject_to(r_1, r_2) \in M$ and $legally_valid(r_1, C_{r_1}) \in S$, then $legally_valid(r_2, C_{r_2}) \notin S$
- (A6) $\forall r_1, r_2 \in R$ if $subject_to(r_1, r_2) \in M$, and $legally_valid(r_1, C_{r_1}) \in S$ and there exists a minimal conflicting set $i \in I$ such that $is_legal(C_{r_1}) \in i$ and $i \subseteq S \cup \{is_legal(C_{r_2})\}$, then $legally_valid(r_2, C_{r_2}) \notin S$.
- (A7) $\forall r \in R$, if $S \models pre_con(r)$, but $legally_valid(r, C_r) \notin S$, then it must be the case that at least one of A4 or A5 or A6 has caused the exclusion of $legally_valid(r, C_r)$. That is if $S \models pre_con(r)$, then unless this would violate one of A5, A6 or A7, it must be the case that $legally_valid(r, C_r) \in S$.
- (A8) S is subset minimal among all sets S' satisfying A1-A7.

5.4 Some remarks on axioms A1-A8

Before we proceed let us give some informal intuition behind some of the axioms and their intended effects. A1 says that all facts in F automatically gain legal status. The set F represents indisputable facts about the legal scenario we are considering. A2 says that if a rule is 'in force' then both the pre-condition and conclusion of that rule must have legal status. Note that it is not enough if

⁷ By $S \models is_legal(pre_con(r))$ we mean that for each positive atom b_i in the conjunction, $is_legal(b_i) \in S$ and for each negated body atom $not\ b_j$ in the conjunction $is_legal(b_j) \notin S$

simply the conclusion has legal status as more than one rule may enforce the same conclusion or the conclusion may be a fact, so we want to know exactly which rules are in force as well as their conclusions. A3 says that anything that has legal status must either be a fact or be a conclusion of some rule that is in force. A4 to A6 describe the semantics of the three modifiers. The intuition for the three modifiers here is that with *despite*, once the precondition of the dominating rule is true, it invalidates the subordinate rule regardless of whether the dominating rule itself is then invalidated by some other rule. With *strong subject to*, once the dominating rule is in force, then it invalidates the subordinate rule. With *subject to*, for the subordinate rule r to be invalidated, it has to be the case that, the dominating rule is in force and, there is a minimal inconsistent set i in I that contains the two atoms in the conclusions of the two rules and, $i \setminus \{C_r\} \subseteq S$. These minimal inconsistent sets along with the *subject to* modifier give us a way to incorporate a classical negation like effect into our system. We will give examples later on to illustrate these modifiers. A7 says essentially that A4-A6 represent the only ways in which a rule whose pre-condition is true may nevertheless be invalidated, and any rule whose precondition is satisfied and is not invalidated directly by some instance of A4-A6, must be in force. A8 says that S must be subset minimal among all sets satisfying A1-A7.

5.5 Non-existence of legal models

Note that there may be configurations for which no legal models exist. This is most easily seen in the case where there is only one rule, the pre-condition of that rule is given as fact, and the rule is strongly subject to itself. There maybe other more non-trivial cases where there are no legal models corresponding to a configuration but so far we haven't been able to formulate explicit examples of this.

5.6 ASP encoding

Here is an ASP encoding scheme given a configuration $Config = (R, F, M, I)$ of legal rules.

```

1 % For any f in F, we have:
2 is_legal(f).
3
4 % Any rule r in R is encoded using the general schema:
5 according_to(r,C_r):-is_legal(pre_con(r)).
6
7 % Say {a,b,c} is a minimal inconsistent set in I, then this
   would get encoded as:
8 opposes(a,b):-is_legal(c)
9 opposes(a,c):-is_legal(b).
10 opposes(b,c):-is_legal(a).
11 %The above is done for every minimal inconsistent set. A pair
    from the set forms the opposes predicate and the rest of
    the elements go in the body

```

```

12
13 % Say {d,e,f,g} is another minimal inconsistent set in I, then
    this would get encoded as:
14
15 opposes(d,e):-is_legal(f),is_legal(g).
16 opposes(d,f):-is_legal(e),is_legal(g).
17 opposes(d,g):-is_legal(f),is_legal(e).
18 opposes(e,f):-is_legal(d),is_legal(g).
19 opposes(e,g):-is_legal(f),is_legal(d).
20 opposes(f,g):-is_legal(d),is_legal(e).
21
22 % If we had a minimal inconsistent set consisting of only 2
    elements say {j,k}, this would get encoded as:
23
24 opposes(j,k).
25
26 % Opposes is a symmetric relation
27 opposes(X,Y):-opposes(Y,X).
28
29
30 % Encoding for 'despite'
31 defeated(R,C,R1):-according_to(R,C),according_to(R1,C1),despite(R,R1).
32
33 %Encoding for 'subject_to'
34 defeated(R,C,R1):-according_to(R,C),legally_valid(R1,C1),opposes(C,C1),subject_to(R1,R).
35
36 % Encoding for 'strong_subject_to'
37 defeated(R,C,R1):-according_to(R,C),legally_valid(R1,C1),strong_subject_to(R1,R).
38
39 not_legally_valid(R):-defeated(R,C,R1).
40
41 legally_valid(R,C):-according_to(R,C),not not_legally_valid(R).
42
43 is_legal(C):-legally_valid(R,C).

```

5.7 Lemma

For a configuration $Config = (R, F, M, I)$, let the above encoding be the program ASP_{Config} . Then given an answer set A_{Config} of ASP_{Config} let $S_{A_{Config}}$ be the set of *is_legal* and *legally_valid* predicates in A_{Config} . Then $S_{A_{Config}}$ is a legal model of $Config$.

Proof See Appendix. \square

5.8 Examples

Let us now give an example to illustrate the various concepts/modifiers discussed above. Consider 4 basic rules:

1. If Bob is wealthy, he must buy a Rolls-Royce.
2. If Bob is wealthy, he must buy a Mercedes.
3. If Bob is wealthy, he may spend up to 2 million dollars on cars.
4. If Bob is extremely wealthy, he may spend up to 10 million dollars on cars.

Suppose we know that the Rolls-Royce and Mercedes together cost more than 2 million but each is individually less than 2 million. We also have that rules 1 and 2 are each subject to rule 3 and despite rule 1, rule 4 holds. Additionally, we also have the fact that Bob is wealthy. In this situation we would expect 2 legal models. One in which exactly rule 1 and rule 3 are legally valid and one in which exactly rule 2 and rule 3 are legally valid. Let us see what our encoding looks like.

```

1 is_legal(wealthy(bob)).
2 % Rules
3 according_to(1,must_buy(rolls,bob)):-is_legal(wealthy(bob)).
4 according_to(2,must_buy(merc,bob)):-is_legal(wealthy(bob)).
5 according_to(3,may_spend_up_to_one_mill(bob)):-is_legal(wealthy(bob)).
6 according_to(4,may_spend_up_to_ten_mill(bob)):-is_legal(extremely_wealthy(bob)).
7
8 % {(must_buy(rolls,bob),must_buy(merc,bob),
9     may_spend_up_to_one_mill(bob)) is a min. inconsistent set.
10
11 opposes(must_buy(rolls,bob),must_buy(merc,bob)):-is_legal(may_spend_up_to_one_mill(bob)).
12 opposes(must_buy(rolls,bob),may_spend_up_to_one_mill(bob)):-is_legal(must_buy(merc,bob)).
13
14 opposes(must_buy(merc,bob),may_spend_up_to_one_mill(bob)):-is_legal(must_buy(rolls,bob)).
15
16 opposes(X,Y):-opposes(Y,X).
17
18 subject_to(3,1).
19 subject_to(3,2).
20 despite(3,4).
21
22 % Encoding for 'despite'
23 defeated(R,C,R1):-according_to(R,C),according_to(R1,C1),despite(R,R1).
24
25 % Encoding for 'subject_to'
26 defeated(R,C,R1):-according_to(R,C),legally_valid(R1,C1),opposes(C,C1),subject_to(R1,R).
27
28 % Encoding for 'strong_subject_to'
29 defeated(R,C,R1):-according_to(R,C),legally_valid(R1,C1),strong_subject_to(R1,R).
30
31 not_legally_valid(R):-defeated(R,C,R1).
32
33 legally_valid(R,C):-according_to(R,C),not not_legally_valid(R).
34
35 is_legal(C):-legally_valid(R,C).

```

Running the the program gives exactly 2 answer sets corresponding to the legal models described above. Now if we add say *strong_subject_to*(3,1) to the set of modifiers then we get exactly one legal model/answer set where exactly rule 3 and rule 2 are legally valid but not rule 1 because it has been invalidated due to rule 3 being legally valid with no regard for the minimal inconsistent sets.

Lastly, if we add *extremely_wealthy*(bob) to the set of facts, then we get a single legal model/answer set where exactly rule 1, rule 2 and rule 4 are legally valid. This is because the rule 3 has been invalidated and hence there are no constraints now on the validity of rule 1 and rule 2.

At this point, we wish to remind the reader that if there was a 5th rule in this rule set and we had a *despite*(4,5) modifier, then as long as the precondition of rule 4 is true, it would still invalidate rule 3 even if rule 4 itself got invalidated by rule 5.

However, in the case of *subject_to* and *strong_subject_to*, the dominating rules needs to be legally valid to invalidate the subordinate rule.

As an illustration of the previous point say we have a fifth rule which says, if Bob owns a company, he may spend up to 20 million dollars on cars, and we have *despite*(4,5) as an additional modifier. Suppose now also we have the three facts that Bob is wealthy, Bob is extremely wealthy and Bob owns a company. Then we would get exactly one legal model/answer set in which exactly rule 1, rule 2 and rule 5 were legally valid. So rule 4 would invalidate rule 3 even though it itself is invalidated by rule 5.

6 Conclusions

Acknowledgements. We are grateful to

This research is supported by the National Research Foundation (NRF), Singapore, under its Industry Alignment Fund – Pre-Positioning Programme, as the Research Programme in Computational Law. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not reflect the views of National Research Foundation, Singapore.

MS: TBD

MS: TBD