Automating Defeasible Reasoning in Law

How Khang Lim, Avishkar Mahajan[nr], Liyana Muthalib[0000−0002−8602−3734], Martin Strecker[0000−0001−9953−9871], and Meng Weng Wong

Singapore Management University

Abstract. The paper studies defeasible reasoning in rule-based systems, in particular about legal norms and contracts. We identify rule modifiers that specify how rules interact and how they can be overridden. We then define rule transformations that eliminate these modifiers, leading in the end to a translation of rules to formulas. For reasoning with and about rules, we contrast two approaches, one in a classical logic with SMT solvers as proof engines, one in a non-monotonic logic with Answer Set Programming solvers.

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 or with a contract 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 project[1]: 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

[1] https://cclaw.smu.edu.sg/

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 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.

> *Remark to the reviewers*: Please read the material in the Appendix at your discretion. It will be removed from the conference version of this paper; the full paper will be made available on `arXiv.org`.

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 There is a huge body of work both on computer-assisted legal reasoning and (not necessarily related) defeasible reasoning. In a seminal work, Sergot and Kowalski [**?**,**?**] code the British Nationality Act in Prolog, exploiting Prolog's negation as failure for default reasoning.

The Catala language [**?**], extensively used for coding tax law and resembling more a high-level programming language than a reasoning formalism, includes default rules, which are however not entirely disambiguated during compile time so that run time exceptions can be raised.

An entirely different approach to tool support is taken with the LogiKEy [**?**] workbench that codes legal reasoning in the Isabelle interactive proof assistant, paving the way for a very expressive formalism. In contrast, we have opted for a DSL with fully automated proofs which are provided by SMT respectively ASP solvers. These do not permit for human intervention in the proof process, which would not be adequate for the user group we target. Symboleo [**?**] and the NAI Suite [**?**] emphasise deontic logic rather than defeasible reasoning (the former is so far not considered in our L4).

As a result of a long series of logics, see for example [?,?], Governatori and colleagues have developed the Turnip system[2] that is based on a combination of defeasible and deontic logic. The system is applied, among others, to modelling traffic rules [?].

It seems vain to attempt an exhaustive review of defeasible reasoning. Before the backdrop of foundational law theory [?], there are sometimes diverging proposals for integrating defeasibility, sometimes opting for non-monotonic logics [?], sometimes taking a more classical stance [?]. Defeasible rule-based reasoning in the context of argumentation theory is discussed in [?,?].

On a more practical side, Answer Set Programming (ASP) [?] goes beyond logic programming and increasingly integrates techniques from constraint solving, such as in the sCASP system [?]. In spite of a convergence of SMT and CASP technologies, there are few attempts to use SMT for ASP, see [?]. For the technologies used in our own implementation, please see Section ??.

MS: find some more representative refs, also in high-level journals

MS: introduce macro?

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 are not relevant for the topic of this paper but are relevant in the future.

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.

For lack of space, we cannot give a full description here – see Appendix ?? for more details. We will illustrate the concepts with an example, a (fictitious) regulation of speed limits for different types of vehicles. Classes are, for example, `Car` and its subclass `SportsCar`, `Day` and `Road`. We will in particular be interested in specifying the maximal speed `maxSp` of a vehicle on a particular day and type of road, and this will be the purpose of the 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. A *formula* is just a Boolean expression, and, consequently, so are quantified formulas `forall x:T. form` and `exists x:T. form`.

[2] https://turnipbox.netlify.com/

In its most complete form, a *rule* is composed of a list of variable declarations introduced by the keyword `for`, a precondition introduced by `if` and a post-condition introduced by `then`. Figure **??** gives an example of rules of our 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 post-conditions are Boolean formulas that can be arbitrarily complex, thus are not limited to conjunctions of literals in the preconditions or atomic formulas in the post-conditions.

```
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. 1: Rules of speed limit example

Rules whose precondition is `true` can be written as `fact` without the `if ... then` keywords. 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` $\overrightarrow{v}$`:` $\overrightarrow{T}$ `if Pre` $\overrightarrow{v}$ `then Post` $\overrightarrow{v}$

corresponds to a universally quantified formula $\forall \overrightarrow{v} : \overrightarrow{T}. \mathit{Pre}\overrightarrow{v} \longrightarrow \mathit{Post}\overrightarrow{v}$ 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 Sections **??** and **??**.

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 super-sort of C and B.

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

Fig. 2: Assertions of speedlimit example

The purpose of our formalization efforts is to be able to make assertions and prove them, such as the statement in Figure **??** which 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.

3 Reasoning with and about 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:
$\{r_1\{\texttt{restrict} : \{\texttt{despite}\ r_2\} \uplus a_1\},\ \ r_2\{\texttt{restrict} : a_2\}, \dots\} \longrightarrow$
$\{r_1\{\texttt{restrict} : a_1\},\ \ r_2\{\texttt{restrict} : \{\texttt{subjectTo}\ r_1\} \uplus a_2\}, \dots\}$

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, ..., `subjectTo` r_n into `subjectTo` $[r_2 \dots r_n]$):
subjectToElim:
$\{r_1\{\texttt{restrict} : \{\texttt{subjectTo}\ [r_2, \dots, r_n]\}\}, \dots\} \longrightarrow$
$\{r_1^o\{\texttt{source}\}, r_1\{\texttt{derived} : \{\texttt{apply} : \{\texttt{restrictSubjectTo}\ r_1^o\ [r_2 \dots r_n]\}\}\}, \dots\}$

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 $\prec_R$ defined by: $r \prec_R r'$ iff r appears in the defining expression of r'. If $\prec_R$ is not a strict partial order (in particular, if it is not cycle-free), then evaluation fails. Otherwise, we order the rules topologically by $\prec_R$ and evaluate the expressions starting from the minimal elements. Obviously, the order $\prec_R$ does not prevent rules from being recursive.

Example 3. It is at this point that the cyclic dependence already remarked after Example **??** will be discovered. We have:
`maxSpCarWorkday'Orig`, `maxSpCarHighway` $\prec_R$ `maxSpCarWorkday`
`maxSpCarHighway'Orig`, `maxSpSportsCar` $\prec_R$ `maxSpCarHighway`
`maxSpSportsCar'Orig`, `maxSpCarWorkday` $\prec_R$ `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:
{ `maxSpSportsCar'Orig` `maxSpCarHighway'Orig`, `maxSpCarWorkday` }
$\prec_R$ `maxSpSportsCar` $\prec_R$ `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 $[] = r_1$
- **`restrictSubjectTo`** r_1 $(r' \uplus rs) =$
  `restrictSubjectTo` $(r_1(\mathit{precond} := \mathit{precond}(r_1) \wedge \neg \mathit{precond}(r')))$ rs

where $\mathit{precond}(r)$ selects the precondition of rule r and $r(\mathit{precond} := p)$ 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 `...` x_n`:`T_n `if Pre(`$x_1,\dots,x_n$`) then 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 `...` y_m`:` S_m `if Pre(`$e_1,\dots,e_n$`) then 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 $\prec_R$, 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`** $r_1\ (r' \uplus rs) =$ **`restrictSubjectTo`** $(r_1(precond := precond(r_1) \wedge \neg postcond(r')))\ rs$
   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. Least fixpoint semantics of logic programs are discussed in [**?**,**?**]. The discussion below 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 $x_1, \ldots, x_n$, 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: $\forall x_1, \ldots, x_n.Pre(x_1, \ldots x_n) \longrightarrow Post(x_1, \ldots, x_n)$.

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

For any rule set $\mathcal{R}$ and predicate P, we can form the set of P-rules, $\mathcal{R}[P]$, as

$$\{\forall x_1, \ldots, x_n.Pre_1[P](x_1, \ldots x_n) \longrightarrow P(x_1, \ldots, x_n), \ldots,$$
$$\forall x_1, \ldots, x_n.Pre_k[P](x_1, \ldots x_n) \longrightarrow P(x_1, \ldots, x_n)\}$$

as the subset of $\mathcal{R}$ containing all rules having P as post-condition. The notation $F[P]$ is meant to indicate that the F can contain P. It can also be taken as a *functional*, *i.e.* a higher-order function having P as parameter.

We say that a functional F is *semantically monotonic* if

$$(\forall x_1, \ldots, x_n.\ P(x_1, \ldots, x_n) \longrightarrow P'(x_1, \ldots, x_n)) \longrightarrow (\forall \overrightarrow{v}.F[P] \longrightarrow F[P'])$$

A sufficient condition for semantic monotonicity is syntactic monotonicity: P does not occur under an odd number of negations in F.

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

$$P^*(x_1, \ldots, x_n) = \forall P.\ (\bigwedge \mathcal{R}[P]) \longrightarrow P(x_1, \ldots x_n)$$

where $\bigwedge \mathcal{R}[P]$ is the conjunction of all the rules in $\mathcal{R}[P]$.

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. By the Knaster-Tarski theorem, P^*, as the least fixpoint of a monotonic functional, is consistent (see a counterexample in Example **??**).

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

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

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 Inv_P

$$\forall x_1, \ldots, x_n.\ P(x_1, \ldots, x_n) \longrightarrow Pre_1(x_1, \ldots x_n) \vee \ldots \vee Pre_k[P](x_1, \ldots x_n)$$

called the *inversion formula of* P, and take it as an approximation of the effect of P^* in Lemma **??**.

As usual, a disjunction over an empty set is taken to be the falsum $\bot$. Assume there are no defining rules for a predicate P, then $Inv_P = P \longrightarrow \bot = \neg P$, which corresponds to a closed-world assumption for P.

Example 8. One motivation for the monotonicity constraint is the following: The simplest example of a rule that is not syntactically monotonic is $\neg P \longrightarrow P$. Its inversion is $P \longrightarrow \neg P$. The two formulas together, $P \leftrightarrow \neg P$, are inconsistent.

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

MS: introduce that notation

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

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 $\texttt{Rulename}_C$, 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 $\forall x_1, \ldots, x_n.\ Pre(x_1, \ldots, x_n) \longrightarrow C(x_1, \ldots, x_n)$ with name rn, whenever the n-tuple $(a_1, \ldots, a_n)$ satsifies the precondition Pre under $\mathcal{M}_P$ and, consequently, $(a_1, \ldots, a_n) \in C^{\mathcal{M}_P}$, we will have $(rn, a_1, \ldots, a_n) \in (C^+)^{\mathcal{M}_D}$.

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 $\prec_R$. Consider a rule $r_P \in \mathcal{F}_P$ with rule name rn_P which by construction has the form $r_p = \forall x_1, \ldots x_n.\ pre^o_P \wedge \neg pre^1_P \wedge \neg pre^k_P \longrightarrow C(x_1, \ldots, x_n)$. We make a case distinction:

- Assume that for arguments $(a_1, \ldots, a_n)$, interpretation $\mathcal{M}_P$ satisfies the precondition $pre^o_P \wedge \neg pre^1_P \wedge \neg pre^k_P$ and thus also the conclusion. In this case, $(rn_P, a_1, \ldots, a_n) \in (C^+)^{\mathcal{M}_D}$, thus satisfying the related rule $r_D \in \mathcal{F}_D$.
- Assume that for arguments $(a_1, \ldots, a_n)$, interpretation $\mathcal{M}_P$ does not satisfy the precondition. Either pre^o_P is not satisfied, leading again to a satisfying assignment of the related rule r_D, or one of the pre^i_P is satisfied.
  In this case, as the rule r^i_P with precondition pre^i_P is strictly smaller than r_P *w.r.t.* $\prec_R$, by induction hypothesis, also the postcondition of r^i_P 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 $\prec_R$. Consider a rule $r_D \in \mathcal{F}_D$ with rule name rn_D which by construction has the form $r_D = \forall x_1, \ldots x_n.\ pre^o_D \wedge \neg post^1_D(rn_1) \wedge \neg post^k_D(rn_k) \longrightarrow C^+(rn_D, x_1, \ldots, x_n)$. Again, we make a case distinction:

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

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 (ASP) 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. Please see the appendix for a brief overview of ASP and references for further reading.

Our work in this section is inspired by [**?**]. Readers will note that there are similarities between our use of predicates such as $according_to$, $defeated$, *opposes* in our ASP encoding, to reason about rules interacting with each other, and similar predicates that the authors of [**?**] use in their work. However our ASP implementation is much more specific to legal reasoning whereas they seek to implement very general logic based r

MS: Typesetting remarks:

– Herbrand: name of a mathematician, therefore capitalized
– for *emphasis*, use the *emph* macro and not *math* mode which looks ugly with ligatures, compare *stuff* and $stuff$ or *configuration* and $configuration$

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 $Concl(r)$.

MS: The question is whether there is any formal difference between rules and facts. Differently said, can facts be considered as rules with precondition True?

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'.

MS: Homogenize writing, for example *subjectTo* instead of *subject_to*

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

MS: not clear

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

MS: Don't quite see the difference between "is legal" and "legally valid"

(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 $S \models is\_legal(C_r)$ [3]
(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 $c = C_r$.

MS: implies?

(A4) $\forall r_i, r_j \in R$, if $despite(r_i, r_j) \in M$ and $S \models is\_legal(pre\_con(r_j))$, then $legally\_valid(r_i, C_{r_i}) \notin S$
(A5) $\forall r_i, r_j \in R$, if $strong\_subject\_to(r_i, r_j) \in M$ and $legally\_valid(r_i, C_{r_i}) \in S$, then $legally\_valid(r_j, C_{r_j}) \notin S$

MS: Shouldn't it be the inverse: if r_2 is not legally valid, then one is entitled to conclude that C_{r_1} is legally valid, provided one can also show the preconditions of r_1?

[3] 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 $is\_legal(b_j) \notin S$

(A6) $\forall r_i, r_j \in R$ if $subject\_to(r_i, r_j) \in M$, and $legally\_valid(r_i, C_{r_i}) \in S$ and there exists a minimal conflicting set $k \in I$ such that $C_{r_i} \in k$ and $C_{r_j} \in k$ and $is\_legal(k \setminus \{C_{r_j}\}) \subseteq S$, then $legally\_valid(r_j, C_{r_j}) \notin S$.[4]

(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 $legally\_valid(r, C_r) \in S$.

5.4 Some remarks on axioms A1-A7

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 simply require that 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 will be discussed next. Firstly, it may help the reader to read $despite(r_i, r_j)$ as 'despite r_i, r_j'. Thus r_i here is the 'subordinate rule' and r_j is the 'dominating' rule. The idea here is that once the precondition of the dominating rule r_j is true, it invalidates the subordinate rule r_i regardless of whether the dominating rule itself is then invalidated by some other rule. For *strong subject to*, the intended reading for $strong_subject_to(r_i, r_j)$ is something like '(strong) subject to r_i, r_j'. Here r_i can be considered the dominating rule and r_j the subordinate one. Once the dominating rule is in force, then it invalidates the subordinate rule. For *subject to*, the intended reading for $subject\_to(r_i, r_j)$ is 'subject to r_i, r_j'. For the subordinate rule r_j to be invalidated, it has to be the case that, the dominating rule r_i is in force and, there is a minimal inconsistent set k in I that contains the two atoms in the conclusions of the two rules and, $i \setminus \{C_{r_j}\} \subseteq S$. 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.

MS: r_2

MS: r_1

MS: r_1

MS: r_2

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

[4] For a set of atoms A, by $is\_legal(A)$, we mean the set $\{is\_legal(a) \mid a \in A\}$

rule is given as fact, and the rule is strongly subject to itself. See the appendix for some further examples of 'pathological' rule configurations.

5.6 ASP encoding

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

```
% For any f in F, we have:
is_legal(f).

% All the modifiers get added as facts like for example:
despite(1,2).

% Any rule r in R is encoded using the general schema:
according_to(r,C_r):-is_legal(pre_con(r)).

% Given a minimal inconsistent set {a_1,a_2,...,a_n}, this
    corresponds to a set of rules:
opposes(a_1,a_2):-is_legal(a_2),is_legal(a_3),...,is_legal(a_n).
opposes(a_1,a_3):-is_legal(a_2),is_legal(a_4)...,is_legal(a_n).
            .
            .
            .
opposes(a_n-1,a_n):-is_legal(a_1),...,is_legal(a_n-2).

% Opposes is a symmetric relation
opposes(X,Y):-opposes(Y,X).


% Encoding for 'despite'
defeated(R,C,R1) :-
   according_to(R,C), according_to(R1,C1), despite(R,R1).

%Encoding for 'subject_to'
defeated(R,C,R1) :-
   according_to(R,C), legally_valid(R1,C1),
   opposes(C,C1), subject_to(R1,R).

% Encoding for 'strong_subject_to'
defeated(R,C,R1) :-
   according_to(R,C), legally_valid(R1,C1),
   strong_subject_to(R1,R).

not_legally_valid(R) :- defeated(R,C,R1).

legally_valid(R,C):-according_to(R,C),not not_legally_valid(R).

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. □

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.

```
is_legal(wealthy(bob)).
% Rules
according_to(1,must_buy(rolls,bob)) :- is_legal(wealthy(bob)).
according_to(2,must_buy(merc,bob)) :- is_legal(wealthy(bob)).
according_to(3,may_spend_up_to_one_mill(bob)) :-
   is_legal(wealthy(bob)).
according_to(4,may_spend_up_to_ten_mill(bob)) :-
   is_legal(extremely_wealthy(bob)).

% {(must_buy(rolls,bob),must_buy(merc,bob),
    may_spend_up_to_one_mill(bob)} is a min. inconsistent set.

opposes(must_buy(rolls,bob),must_buy(merc,bob)) :-
   is_legal(may_spend_up_to_one_mill(bob)).

opposes(must_buy(rolls,bob),may_spend_up_to_one_mill(bob)) :-
   is_legal(must_buy(merc,bob)).

opposes(must_buy(merc,bob),may_spend_up_to_one_mill(bob)) :-
   is_legal(must_buy(rolls,bob)).

opposes(X,Y):-opposes(Y,X).

```

```
subject_to(3,1).
subject_to(3,2).
despite(3,4).

% Encoding for 'despite'
defeated(R,C,R1) :-
   according_to(R,C),according_to(R1,C1),despite(R,R1).

% Encoding for 'subject_to'
defeated(R,C,R1) :-
   according_to(R,C), legally_valid(R1,C1),
   opposes(C,C1), subject_to(R1,R).

% Encoding for 'strong_subject_to'
defeated(R,C,R1) :-
   according_to(R,C), legally_valid(R1,C1),
   strong_subject_to(R1,R).

not_legally_valid(R) :- defeated(R,C,R1).

legally_valid(R,C) :-
   according_to(R,C), not not_legally_valid(R).

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

This paper has discussed different approaches for representing defeasibility as used in law texts, by annotating rules with modifiers that explicate their relation to other rules. We have notably presented two encodings in classical logic (Sections **??** and **??**) and explained how they are related (Section **??**). Quite a different approach, based on Answer Set Programming, is presented in Section **??**.

An experimental implementation of the L4 ecosystem is under way[5], but it has not yet reached a stable, user-friendly status. It is implemented in Haskell, features an IDE based on VS Code and natural language processing via Grammatical Framework[?]. Currently, only the coding of Sections **??** has been implemented – the coding of Section **??** would require a much more profound transformation of rules and assertions. Likewise, the ASP coding of Section **??** has only been done manually. The interaction with SMT solvers is done through an SMT-LIB [?] interface, thus opening the possibility to interact with a wide range of solvers. As our rules typically contain quantification, reasoning with quantifiers is crucial, and the best support currently seems to be provided by Z3 [?].

We are still at the beginning of the journey. A theoretical comparison of the classical and ASP approaches presented here still has to be carried out, and it has to be propped up by an empirical evaluation. For this purpose, we are currently in the process of coding some real-life law texts in L4. We are fully aware of shortcomings of the current L4, which will strongly evolve in the next months, to include reasoning about deontics and about temporal relations. Integrating these aspects will not be easy, and this is one reason for not committing prematurely on one particular logical framework.

Acknowledgements. We are grateful to

MS: TBD

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.

[5] https://github.com/smucclaw/baby-l4

A An Overview of the L4 Language: Details

Let us give some more details about the L4 language: its class and type definition mechanism, and the way it handles proof obligations.

A.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. 3: 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 entitities, 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: Vehicle`, the expression `v.weight` is equivalent to `weight(v)`, at least logically, even though the operational interpretations may differ.

A.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 $T_1, T_2, \ldots T_n$ are types, then so are function types T_1 `->` T_2 and tuple types `(`T_1`, ... ,`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. 4: 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 `is`C`:` S `-> 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.

MS: Promise

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 $T_1, \ldots, T_n, T'_1, \ldots, T'_n$, if $T_1 \preceq T'_1, \ldots, T_n \preceq T'_n$, then T'_1 `->` $T_2 \preceq T_1$ `->` T'_2 and `(`T_1`, ... ,` T_n`)` $\preceq$ `(`T'_1`, ... ,` 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 $f : A'\text{->}B$ and $a : A$ and $A \preceq A'$, then function application is well-typed with $f\ a : B$.

A.3 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 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 an essential consistency requirement and a rule system violating it is inconsistent *w.r.t.* the intended semantics of `maxSp`. One remedial action is to declare one of the rules as default and the other rule as overrriding 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

B Brief outline of ASP

First we shall give a brief overview of Answer Set Programming. 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:

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

Here h and b_1...,b_n are atoms. For an atom b_i, $not\ b_i$ is the negated atom where the *not* represents negation as failure. Informally $not\ b_i$ 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 $b_1, b_2.., b_k, not\ b_{k+1}..., not\ b_n$ are all true, h is true. h is also sometimes known as the head of the rule and the positive and negated atoms $b_1, b_2.., b_k, not\ b_{k+1}..., not\ b_n$ 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 constrains, 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.

C ASP encoding

Here we recap the ASP encoding scheme given a configuration $Config = (R, F, M, I)$ of legal rules. We will refer to this in the proof of lemma in 5.7, which will be given next.

```
% For any f in F, we have:
is_legal(f).

% All the modifiers get added as facts like for example:
despite(1,2).
subject_to(4,5).

% Any rule r in R is encoded using the general schema:
according_to(r,C_r):-is_legal(pre_con(r)).

% Say {a,b,c} is a minimal inconsistent set in I, then this
    would get encoded as:
opposes(a,b) :- is_legal(c)
opposes(a,c) :- is_legal(b).
opposes(b,c) :- is_legal(a).
%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

% Say {d,e,f,g} is another minimal inconsistent set in I, then
    this would get encoded as:

opposes(d,e) :- is_legal(f),is_legal(g).
opposes(d,f) :- is_legal(e),is_legal(g).
opposes(d,g) :- is_legal(f),is_legal(e).
opposes(e,f) :- is_legal(d),is_legal(g).
opposes(e,g) :- is_legal(f),is_legal(d).
opposes(f,g) :- is_legal(d),is_legal(e).

```

```
% If we had a minimal inconsistent set consisting of only 2
    elements say {j,k}, this would get encoded as:

opposes(j,k).

% Opposes is a symmetric relation
opposes(X,Y):-opposes(Y,X).


% Encoding for 'despite'
defeated(R,C,R1) :-
   according_to(R,C), according_to(R1,C1), despite(R,R1).

%Encoding for 'subject_to'
defeated(R,C,R1) :-
   according_to(R,C), legally_valid(R1,C1),
   opposes(C,C1), subject_to(R1,R).

% Encoding for 'strong_subject_to'
defeated(R,C,R1) :-
   according_to(R,C), legally_valid(R1,C1),
   strong_subject_to(R1,R).

not_legally_valid(R) :- defeated(R,C,R1).

legally_valid(R,C):-according_to(R,C),not not_legally_valid(R).

is_legal(C):-legally_valid(R,C).
```

D Proofs

Proof of the Lemma in 5.7
Firstly note, that the converse of the lemma is false. That is, there are configurations and legal models of those configurations that do not correspond to any answer set of the ASP encoding. A simple example of this is the following: Consider the configuration where there are only 2 rules:
(1): $a \rightarrow a$
(2): $not\ a \rightarrow b$

There are no other facts, modifiers or minimal inconsistent sets. Then for this configuration $\{legally\_valid(1,a), is\_legal(a)\}$ is a legal model but it does not correspond to any answer set of the ASP encoding. As explained in [**?**], this is essentially due to the fact that not all minimal supported models of a logic program are stable models. See [**?**] for the example given above and a further discussion on this topic. Now we shall proceed to the proof of the lemma.

Given a configuration $Config$, let A_{Config} be an answer set of it's ASP encoding and let $S_{A_{Config}}$ be the set of is_legal and $legally\_valid$ predicates in A_{Config}. It is easy to see that A_{Config} satisfies A1-A5. For example if the set M from $Config$ contains $strong\_subject\_to(r_i, r_j)$, then A_{config} would contain $strong\_subject\_to(r_i, r_j)$. Now if $S_{A_{Config}}$ contains $legally\_valid(r_i, C_{r_i})$, then so would A_{Config}. Now, if $pre\_con(r_j)$ is satisfied in A_{Config}, then $according\_to(r_j, C_{r_j})$ is in A_{Config} and therefore $defeated(r_j, C_{r_j}, r_i)$ is in A_{Config} by line 37 of the general encoding shown in 5.6. Therefore $not\_legally\_valid(r_j)$ is in A_{Config} by line 41 of the encoding. Therefore by the line 43 of the encoding, $legally\_valid(r_j, C_{r_j})$ is not in A_{Config}. Therefore $legally\_valid(r_j, C_{r_j})$ is not in $S_{A_{Config}}$.

Now if $pre\_con(r_j)$ is not satisfied in A_{Config}, then $according\_to(r_j, C_{r_j})$ is not in A_{Config} and so again $legally\_valid(r_j, C_{r_j})$ is not in A_{Config} and therefore not in $S_{A_{Config}}$.

We shall now show that $S_{A_{Config}}$ satisfies A6 and A7.

Say the set M contains $subject\_to(r_i, r_j)$ and $legally\_valid(r_i, C_{r_i})$ is in $S_{A_{Config}}$. Furthermore suppose that there exists some $k \in I$ which contains C_{r_i} and C_{r_j} such that $is\_legal(k \setminus \{C_{r_j}\}) \subseteq S_{A_{Config}}$. Then it follows that, $is\_legal(k \setminus \{C_{r_j}\}) \subseteq A_{Config}$. Therefore due to the way that the *opposes* predicates are defined in the encoding, it follows that $opposes(C_{r_i}, C_{r_j})$ is in A_{Config} Now if $pre\_con(r_j)$ is in A_{Config} then it follows from line 31 of the encoding that, $defeated(r_j, C_{r_j}, r_i)$ is in A_{Config}, therefore $legally\_valid(r_j, C_{r_j})$ is not in A_{Config} and therefore not in $S_{A_{Config}}$.

Again as before, if $pre\_con(r_j)$ is not in A_{Config} then $legally\_valid(r_j, C_{r_j})$ is not in A_{Config} and therefore not in $S_{A_{Config}}$.

Suppose $S_{A_{Config}} \models pre\_con(r_j)$, then A_{Config} satisfies $pre\_con(r_j)$. So $according\_to(r_j, C_j)$ is in A_{Config}, then if $legally\_valid(r_j, C_j)$ is not in A_{Config}, according to lines 41 and 43 of the encoding it must be the case that $defeated(r_j, C_j, r_k)$ is in A_{Config} for some rule r_k. But then because of the way that the $defeated$ predicate is defined in lines 28, 32, 37, it must mean that rule r_j is invalidated in accordance with either A4, A5 or A6. So $S_{A_{Config}}$ satisfies A7. □

E Pathological rule configuration examples

In this section we shall briefly give some examples of rule configurations that fail to satisfy certain properties.

One may suspect that given any configuration, the ASP encoding only generates answer sets corresponding to subset minimal legal models. However this is not the case. Consider the configuration where there are 3 rules:
(1) $a \rightarrow c$

(2) $not\ c \to e$
(3) $a \to a$
The only fact is $is\_legal(a)$, and there are 2 modifiers $despite(1,2)$, $strongly\_subject\_to(3,2)$. There are no minimal inconsistent sets.

For this configuration, the ASP encoding generates two answer sets corresponding to the legal models:

$\{is\_legal(a), legally\_valid(3,a)\}$ and $\{is\_legal(a), legally\_valid(3,a), is\_legal(c), legally\_valid(1,c)\}$.

We suspect that the ASP encoding does only return subset minimal legal models if there is no negation as failure in rule pre-conditions or if there are no *despite* modifiers, however pursuing this matter fully is left for future work.

Here we will give an example of a rule configuration that has no legal models even though none of the rule modifiers involve a rule directly being subject to itself.

Consider the configuration where there are 2 rules:
(1) $a \to b$
(2) $b \to c$
The only fact is $is\_legal(a)$, there is one modifier $subject_to(2,1)$ and there is one minimal inconsistent set $\{b,c\}$. Then this rule configuration has no legal models.

F Reasoning with and about Rules

F.1 Facets of Defeasible Reasoning

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 the rest of this section will be dedicated to giving them a semantics in a classical first-order logic, by means of apparently different interpretations. A discussion of a non-monotonic logic, in the context of on Answer Set Programming, will be deferred to (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*. We finish with a comparison of the two strands of reasoning in Section **??**.

MS: as/by??

MS: This intro is a mess and has to be rewritten depending on how we organize the paper

F.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 practictioner *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 explicitating 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 sollicitor[6] 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 explicitating 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

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

- 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

- "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
```

MS: Problem with spaces in lstlisting

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
```

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