# CLAR submission

NN1[1], NN2[2][nr], and Martin Strecker[3][0000−0001−9953−9871]

[1] Foo, SG
[2] Bar, SG
[3] Baz, SG

**Abstract.**

**Keywords:** TO BE ADAPTED Automated Theorem Proving, Modal Logic, Graph Transformations, Program Verification

## 1 Introduction

Related work:

- For bibliographical references, see [4]
- [?,?]
- [?]
- [?]

## 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 [] that are not relevant for the topic of this paper.

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 a running example, a (fictitious) reglementation of speed limits for different types of vehicles.

```
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

### 2.1 Terminology and Class Definitions

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

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.

### 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

[4] https://law.stanford.edu/publications/developing-a-legal-specification-protocol-techno

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. 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 `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 $f\ a : B$.

### 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

MS: scenario changed wrt repo

postcondition 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 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.

```
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` $v_1$`:` $T_1$ `...` $v_n$`:` $T_n$ `P` $v_1 \ldots v_n$

Rules may not contain free variables, so all variables occuring 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 $\forall \overrightarrow{v} : \overrightarrow{T}.\ Pre\overrightarrow{v} \longrightarrow Post\overrightarrow{v}$ 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 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 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

# 3 Defeasible Reasoning

## 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 we 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 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[5] 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

- 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".

We will subsequently come back to this difference.

MS: where?

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

[5] 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:

MS: Problem with spaces in lstlisting

```
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.

### 3.3 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 ??).

**3.3.1 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 $\preceq_R$ 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:
{ `maxSpSportsCar'Orig` `maxSpCarHighway'Orig`, `maxSpCarWorkday` }
$\preceq_R$ `maxSpSportsCar` $\preceq_R$ `maxSpCarHighway`
and will use this order for rule elaboration.

**3.3.2 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` $(r_1(precond := precond(r_1) \wedge \neg precond(r'))) \; rs$

where $precond(r)$ selects the precondition of rule $r$ and $r(precond := p)$ 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(`$x_1,\ldots,x_n$`) then Post(`$x_1,\ldots,x_n$`)`

is remapped with

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

to the rule

`rule <r> for` $y_1$`:` $S_1$ `...` $y_m$`:` $S_m$ `if Pre(`$e_1,\ldots,e_n$`) then Post(`$e_1,\ldots,e_n$`)`

Here, $e_1, \ldots, e_n$ are expressions that have to be well-typed with types $E_1, \ldots, E_n$ in context $y_1 : S_1, \ldots, y_m : S_m$ (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))
  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)`.

**3.3.3 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, $\texttt{C1}^+$`: 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
```

### 3.4 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 $x_1, \ldots, x_n$, 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 $P(\dots e \dots) = (\forall x. x = e \longrightarrow P(\dots x \dots))$ 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

$$\begin{aligned} P^*(x_1, \dots, x_n) = \forall P.\ & (Pre_1(x_1, \dots x_n) \longrightarrow P(x_1, \dots, x_n)) \longrightarrow \dots \\ & (Pre_k(x_1, \dots x_n) \longrightarrow P(x_1, \dots, x_n)) \longrightarrow \\ & P(x_1, \dots, x_n) \end{aligned}$$

$P^*$ 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 $P^*(x_1, \dots, x_n) = \forall P.\ P(x_1, \dots, x_n) = \bot$.

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 $P$.

### 3.5 Answer Set Programming

MS: section probably does not have an adequate name

### 3.6 Comparison

## 4 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 first begin by formalizing the notion of what it means to 'satisfy' a rule set. We will do this in a way that is most amenable to ASP.

### 4.1 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 it's conclusion $Concl(r)$

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 $r$ in $R$ enforces conclusion $c$. The unary predicate $is\_legal(c)$ intuitively means that $c$ legally holds. The predicates $despite$, $subject\_to$ and $strong\_subject\_to$ all cause some rules to override others. Their precise semantics will be given next.

### 4.2 Semantics

A set $S$ of $is\_legal$ and $legally\_valid$ predicates is called a $model$ 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 $S \models is\_legal(C_r)$ (this is a slight abuse of notation. Explain later?)

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 $is\_legal(C_{r_1}) \in i$, $i \not\subseteq S$, 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 instance of A4 or A5 or A6 holds in which $r$ is the 'lower priority rule'. (Maybe explicitly state what this means?) .

### 4.3 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 conclusion is enforced by a rule then both the pre-condition and conclusion of that rule must have legal status. Note that it is not enough if 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 enforced by the rules. 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 higher priority rule is true, it invalidates the lower priority rule regardless of whether the higher priority rule actually comes into effect. With 'strong subject to', once the higher priority rule is in effect, then it invalidates the lower priority rule. With 'subject to', The higher priority rule has to be in effect and it has to contradict the lower priority rule for the lower priority rule to be invalidated. 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 not be in effect, and any rule whose precondition is satisfied and is not invalidated directly by some instance of A4-A6 must be in effect.

## 5 Non-existence of models

Note that there may be configurations for which no 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.

## 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).

% 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 form
%predicate and the rest of the elements go in the body.

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


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

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

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

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

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

### 6.1 Lemma

For a configuration $Config = (R, F, M, I)$, let the above encoding be the program $ASP_{Config}$. Then the sets of $is\_legal$ and $legally\_valid$ predicates from answer sets of $ASP_{Config}$ correspond exactly to the *models* of $Config$.

*Proof sketch* The definition of $defeated(R, C)$ that involves *despite* encodes exactly A4, the definition of $defeated(R, C)$ that involves $strong\_subject\_to$ encodes exactly A5, and the way the *opposes* predicates are defined together with the remaining definition of $defeated(R, C)$ encodes exactly A6. Other than rules that are invalidated this way, all $according\_to(R, C)$ predicates get turned into $legally\_valid(R, C)$ predicates (A7), and the only way to derive a $is\_legal(C)$ predicate is if $C$ is a fact or $C$ has been enforced by a rule. (First and last lines of encoding)

### 6.2 Comments/Thoughts

It is not entirely clear to me how to unify this approach with Martin's approach to the semantics of L4/where this stuff fits in exactly. However I think it can be done? It seems that our notions of what a rule is match well. Class hierarchies can be incorporated using rules like $is\_truck(X) \rightarrow is\_vehicle(X)$. In this set-up this information along with minimal inconsistent sets can all be stored in the parameter $I$ from the tuple. So perhaps we could rename it $T$ for background theory?

The minimal inconsistent sets in $I$ seem to somewhat loosely correspond to 'assertions' in Martin's set up although they seem to mean different things. In this set up, they are meant to encode basic domain knowledge about things that contradict each other. But I think they can also be used as a tool to check properties of the program.

Our notions of 'subject to' are different but 'strong subject to' here seems close to Martin's 'subject to'.

I'm not sure how to go about properly proving the lemma.

Need to do examples to illustrate the different kinds of modifiers.

## 7 Conclusions

*Acknowledgements.* We are grateful to