We describe a reasoning and unification algorithm over arbitrary terms and rules, by showing that the problem of backtracking a query can be transformed into a similar reasoning problem over one pre-order, one equivalence relation, rules, no variables, and no lists.

More concretely, given an input of the form:

1. Term := Atom|Var|(Term*)
2. Rule := (Term*) => Term

Reads: a term is an arbitrary tree of atoms and variables. Our assumed semantics is that atoms are all different and may not unify with each other. Variables may unify with any term. Rules have the form of a set of terms implying a consequence when all are met.

Our target language has the form:

1. Term := Sym<Sym|Sym∼Sym
2. Rule := (Sym*)=> (Sym*)

means, given a finite set of symbols, a term is either stating an equivalence between them, or a partial order. Note that unlike the input language, terms may not be nested and always consist of three parts.

The rules in the target language reason only over the equivalence relation. They are a translation of (and therefore depend on) the input rules. But the target language has to contain a few more constant rules, in which reasoning over the translated input together with these rules is equivalent to reasoning over the input language using sequent calculus and unification.

The *suffix* set of a symbol x is the set of all y such that $x < y$. Let x, y be any two symbols. The rules of the target languge can be summarized into three:

1. If $x < y$ then $\neg (x \sim y)$.
2. Suffixes of equivalent symbols are equivalent.
3. $<$ is a partial order and $\sim$ is an equivalence relation.

To hint the transformation from the input language to the target language: symbols are coordinates in term's tree. For example, the coordinate of c in (a b (c)) is (3,1). The symbols contain a little more information, by prefixing evrey coordinate with head/body numbers, so a symbol points to a certain location in the kb. Those coordinates are therefore lists of integers. If given two such lists and one is a prefix of the other, it means that the prefix coordinate represent a list in the input language. Suffixes are the list's nested elements. The requirement of equivalent lists to have equivalent suffixes has to do with unifying lists requiring unifying all their elements.

An element with an empty suffix is necessarily not a list. We make that sure by treating rdf:nil as a literal. In the target language, the relation $<$ between symbols reflects with coordinates are prefix of each other (though we don't need to keep the coordinates themselves, only the relation). The relation $\sim$ reflects same coordinates sharing same value. For example, if two coordinates point to different location on the kb in which they both happen to contain rdf:nil, then they will have to be stated as equivalent.

The explicit transformation is as follows:

1. Define a symbol per every location of term/subterm in the source kb. The source kb is a list of rules. The symbols should treat equal terms on different

rules/heads/bodies as different. So every symbol points to a unique place in the kb.

2. Given two terms and two symbols that represent them, then one is $<$ the other iff the former is a strict subterm of the latter.

3. Given two symbols, then they are stated as equivalent iff the atom/var they point to is the same one. Variables are treated at the rule's level.

4. The target rules are a direct translation of the source rules, replacing every term by the coordinates of its beginning. Matching rules though is done via equivalence. Modus ponens in the target language is:

$$\frac{\begin{array}{c} x \\ x \sim y \\ y \implies z \end{array}}{z}$$

together with the additional above judgements, formalized:

$$\frac{\begin{array}{c} x < y \\ x \sim y \end{array}}{\bot} \qquad \frac{\begin{array}{c} x < y \\ y < x \end{array}}{\bot} \qquad \frac{}{x \sim x} \qquad \frac{\begin{array}{c} x \sim y \\ y \sim z \end{array}}{x \sim z} \qquad \frac{\begin{array}{c} x < y \\ y < z \end{array}}{x < z} \qquad \frac{\begin{array}{c} x \sim y \\ x < z \end{array}}{y < z}$$

Tedious simplification yields the only rule needed in the target language:

$$(x \not< y) \vee ((y \not< x) \wedge (x \nsim y))$$

one can easily verify that this formula encapsulate everything said on 1,2,3 in page 1.