

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 language 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 every 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:

$$\frac{x < y}{x \sim y} \quad \frac{x < y}{y < x} \quad \frac{}{x \sim x} \quad \frac{x \sim y}{y \sim z} \quad \frac{x < y}{y < z} \quad \frac{x \sim y}{x < z}$$

or

$$\begin{array}{l} \neg(x < y \wedge x \sim y) \quad \wedge \\ \neg(x < y \wedge y < x) \quad \wedge \\ x \sim x \quad \wedge \\ (x \sim z) \vee \neg(x \sim y \wedge y \sim z) \quad \wedge \\ (x < z) \vee \neg(x < y \wedge y < z) \quad \wedge \\ (y < z) \vee \neg(x \sim y \wedge x < z) \quad \wedge \\ (\neg x \vee z) \vee \neg(x \sim y \wedge (z \vee \neg y)) \end{array}$$

(the last one is our modus ponens written as $\frac{x \sim y}{y \implies z}$). Pushing negations inside:

$$\begin{array}{l} (x \not< y) \vee (x \approx y) \quad \wedge \\ (x \not< y) \vee (y \not< x) \quad \wedge \\ (x \sim x) \quad \wedge \\ (x \sim z) \vee (x \approx y) \vee (y \approx z) \quad \wedge \\ (x < z) \vee (x \not< y) \vee (y \not< z) \quad \wedge \\ y < z \vee x \approx y \vee x \not< z \quad \wedge \\ \neg x \vee z \vee x \approx y \vee (z \wedge \neg y) \end{array}$$

where the last line can be simplified further into $\neg x \vee z \vee x \approx y$.

On this representation x, y, z run over all permutations of all coordinates. For kb with n coordinates (which is roughly a little less the size of the kb itself) we therefore generated a 3-CNF with 7 clauses per possible unordered choice of x, y, z . The number of such choices is $\frac{n(n-1)(n-3)}{6}$, ending up with $\frac{7}{6}n(n-1)(n-3)$ 3-CNF (implicit) clauses only for the constant rules, as a function of n . The more clauses are given explicitly in 1-CNF in the form of $x < y, x \sim y$. They could therefore be virtually substituted into the implicit 3-CNF and reducing the search space. Note that the explicit clauses are typically $\mathcal{O}(n^2)$ in number, as they speak about pairs of kb coordinates. Lastly, the rules themselves in the source language need to be restated in the target language, but they rather take shorter space as they keep only the coordinates of the root terms.

We can further simplify the above 7 clauses into the negation of

$$\begin{aligned} & (x < y) \wedge ((y < x) \vee (x \sim y)) \quad \vee \\ & [(x \not< z) \wedge (x < y)] \neq [(x \sim y) \wedge (x < z)] \quad \vee \\ & (x \sim y) \wedge [(x \wedge \neg z \wedge (\neg z \vee y)) \vee [(x \approx z) \wedge (y \sim z)]] \quad \vee \\ & (x \approx x) \end{aligned}$$

given we implement the equivalence relation with union-find, so we shouldn't care about reflexivity and transitivity of equivalence, we can write the DNF as:

$$\begin{aligned} & (x < y) \wedge ((y < x) \vee (x \sim y)) \quad \vee \\ & [(x \not< z) \wedge (x < y)] \neq [(x \sim y) \wedge (x < z)] \quad \vee \\ & (x \sim y) \wedge x \wedge \neg z \wedge (\neg z \vee y) \end{aligned}$$

where the last one is our special modus ponens. Therefore we can merely backtrack the target language using union-find-implemented equivalence rather unification, and assuming trivial that it cannot be that $x < y$ and $x \not< y$, while refutations are merely in cases of:

$$(x < y) \wedge ((y < x) \vee (x \sim y)) \vee [(x \not< z) \wedge (x < y)] \neq [(x \sim y) \wedge (x < z)]$$

trying to simplify further:

$$\begin{cases} x \not< z & (x < y) \wedge ((y < x) \vee (x \sim y)) \\ x < z & (x < y) \wedge ((y < x) \vee (x \sim y)) \end{cases}$$

we see that we get exactly same result:

$$(x < y) \wedge ((y < x) \vee (x \sim y))$$

is the only refutation needed if doing equivalence relation with union-find, and backtracking with merely equivalence rather unification. Inverting back we get the only rule needed in the target language:

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

which means: requiring \sim to be a well defined equivalence, and $<$ to be a well defined partial order, and requiring that $x < y$ will always imply $x \approx y$, encapsulates all the unification and reasoning process.