Efficient Rule-Matching for Hyper-Tableaux

Bjarne Holen
Department of Informatics, University of Oslo, Norway
Dag Hovland
Department of Informatics, University of Oslo, Norway
Martin Giese
Department of Informatics, University of Oslo, Norway

Abstract

Over the past decades, a number of calculi for automated reasoning have been proposed that share some core features: 1. proofs are built in a tableau/sequent style as trees where nodes are labeled with literals, and 2. these proofs are expanded by interpreting the problem clause set as a set of rules, and requiring all negative literals in clauses to present on a branch for expansion. This applies to hyper-tableaux [1], MGTP [7], coherent logic [3, 4], and others. Existing implementations typically spend much of their time in the process of matching branch literals with the negative literals of the input clauses. We present an alternative to this matching process by applying a modified version of the *RETE algorithm* [6]. The RETE algorithm was developed in the 1970s for *production systems* in artificial intelligence. We exploit the similarities between the mentioned calculi and production systems in order to make the RETE algorithm solve the matching problem. We also investigate the effect of working on several independent branches present in tableau proof search but not in production systems.

1 Introduction

The goal of this paper is to illustrate how the RETE algorithm [6] can be modified to fit the matching problem of different first order procedures which share some characteristics. Proof procedures that match an expanding set of atoms against a static set of negative clause literals must have a way to store partial matches in order to redo as little as possible as the fact-set expands. Literals that occur in multiple clauses can be exploited to minimize repeated matching. The RETE algorithm handles many of these issues, although for a different problem domain; production systems. Similarities between production systems and these logical calculi will be illustrated, and we will show how the differences can be overcome in order to adapt the RETE algorithm to build an automated reasoner. A type of Hyper-tableaux [1] referred to as Coherent Logic (CL) [3] or Geometric Logic [4], will be used to illustrate the process.

1.1 Coherent Logic

Coherent Logic is a fragment of First Order Logic (FOL), where only closed formulas of the following form are allowed:

$$\forall \vec{X}\ [A_1 \wedge \ldots \wedge A_n \ \longrightarrow\ \exists \vec{Y}(\ C_1 \vee \ldots \vee\ C_k)]$$

The A_i's are atomic formulas (predicates applied to a list of terms) and the C_i's are conjunctions of atomic formulas. $\vec{X}$ and $\vec{Y}$ are non-overlapping lists of variable names (without repetition). The left hand sides of coherent formulas are sometimes referred to as the antecedent or the argument, and the right hand sides as the succedent or conclusion. A coherent theory consists of a set of coherent formulas, where each formula is referred to as an *axiom* or a *rule*. As usual, a term is a constant, a variable, or a function applied to one or more terms. A formula without

free variables is referred to as a closed formula, a formula without variables is referred to as a ground formula, and an atomic ground formula is referred to as a *fact*. The set of free variables occurring in a formula Φ, is denoted $\mathsf{vars}(\Phi)$. We use capital letters for variables (X, Y, Z), lowercase letters for constants (a, b, c), functions (f, g, h), and predicates (p, q, r). The formulas considered in this paper are all coherent formulas. There exists translations taking arbitrary FOL formulas to equisatisfiable CL theories [3, 4].

The scope of the universal quantifier(s) is the entire formula; existential quantifiers are limited to the conclusion. Both sides of the implication can be empty; an empty left hand side is represented by $\top$ (since it is implicitly true), whereas an empty right hand side is represented by $\bot$ (since it is implicitly false). Fig. 1 shows an example of a coherent theory.

As is common, we will leave the universal quantifiers implicit, and only write the existential quantifiers. Furthermore, we do not allow *rigid* universal variables, that is, universal variables not occurring in the left hand side, but occurring in more than one disjunct on the right hand side. It is well-known that any set of CL formulas can easily be transformed into an equisatisfiable set with no rigid variables, by the introduction of a domain predicate.

$$\top \longrightarrow p(a) \tag{1}$$
$$p(X) \longrightarrow z(X, X) \tag{2}$$
$$z(X, X) \longrightarrow \exists Y, Z : q(X, Y) \vee q(X, Z) \tag{3}$$
$$q(X, Y) \longrightarrow q(Y, X) \tag{4}$$
$$q(X, Y) \wedge q(Y, X) \longrightarrow \bot \tag{5}$$

Figure 1: A coherent theory

The RETE algorithm has been successfully used for other types of reasoning in the past, in [9, 11] they optimize the process of hyper-linking [8]. This is an instance-based method in which first order formulas are instantiated in order to make them propositional, and a regular propositional SAT-solver is used in rounds.

This paper is organized in the following sections. Section 2 gives a quick overview of production systems, section 3 shows the proof procedure/calculus, and section 4 explains the basics of the RETE algorithm. Section 5 presents optimizations and modifications of the algorithm, and section 6 displays some results. Conclusion and future work is in section 7.

2 Production Systems

An overview of a production systems will be given in this section to illustrate their characteristics. A *production* can be seen as a set of requirements, with a belonging *action*, typically presented as in 6.

$$R_1 \wedge R_2 \wedge \cdots \wedge R_n \;\rightarrow\; A \tag{6}$$

The intuition is that fulfilling the requirements $(R_1, \ldots, R_n)$ justifies the action. A typical production system will have more than one production, referred to as *production-rules*. The term *working memory* relates to our current knowledge, in the sense that knowledge is what is

needed in order to satisfy *requirements*, i.e. production rules are satisfied based on knowledge from our working memory. Performing an *action* typically generates new knowledge which is added to the working memory. A production system can be as simple as a propositional consequence relation, like the one shown in Fig. 2.

$$p, q \Rightarrow r \tag{7}$$
$$r, p \Rightarrow t \tag{8}$$
$$t, q \Rightarrow \mathit{goal} \tag{9}$$

Figure 2: Production System

All we need to fulfill requirements in a consequence relation, is propositions inside our working memory satisfying the argument of a production. Assume that we initially have these elements: $\{p, q\}$, inside our working memory. We can match the production-rules requirements with the elements inside the working memory, Fig. 3 displays the situation after the first round of matching.

$$\mathbf{p}, \mathbf{q} \Rightarrow r \tag{10}$$
$$r, \mathbf{p} \Rightarrow t \tag{11}$$
$$t, \mathbf{q} \Rightarrow \mathit{goal} \tag{12}$$

Figure 3: After 1 Round of Matching

The requirements written in **bold** are fulfilled, and the first production rule can be applied. The first rule's action adds the element: r to our working memory. Matching our new element r against the production rules, fulfills the requirements of the second rule: $\mathbf{r}, \mathbf{p} \Rightarrow t$, and its action can be preformed, resulting in the new element: t. The term t fulfills the last requirement of the third production-rule: $\mathbf{t}, \mathbf{q} \Rightarrow \mathit{goal}$. Assuming that we were trying to reach the *goal*-term, we have succeeded. From this small example, some of the problems that the RETE algorithm addresses become clear. *Static production rules* implies that we only have to process the working memory elements once, i.e., they will always match the same elements. *Production rules can share requirements*, which can be pre-processed to fulfill them all in one step. The process of *matching* (fulfilling requirements) will typically be more complex for a real production system.

3 Automating Coherent Logic

We base our proof procedure on a type of ground forward chaining [3, 2]. We start out with an initial set of facts containing closed first order predicates. The initial fact-set is constructed from a set of atoms which are true based on assumption, as implied by the axioms with empty left-hand sides. (E.g. the first axiom in the coherent theory presented in Fig. 1.) The axioms of a coherent theory play the role of rules in a production system. An axiom/rule can be applied if one or more elements from the fact-set match the left hand side, (argument/antecedent) of the rule. Matching in this context refers to a regular first order unification [10], where variables can be substituted for terms in order to make them syntactically identical. We use

the term *matching* to indicate that our fact-set contains closed terms or predicates, i.e. a simple unification scheme can be used, since there are no shared values.

A match can lead to one or more substitutions that can be applied on the right hand side of the implication in order to generate new facts.

Disjunctions on the right hand side of a rule or conclusion lead to forks in the proof tree, where each disjunction becomes its own branch, that has to be closed individually. All the branches contain the same set of facts as we branch out, and from that point on they extend the fact-set in in different ways.

Now we will give a formal definition of the proof search, after introducing some notation.

Definition 3.1 (Substitution and Instances)**.** *A (ground)* substitution *is a mapping from variables to (ground) terms.* $\{X_1 \leftarrow t_1, \ldots, X_n \leftarrow t_n\}$ *denotes the substitution mapping variable* X_i *to term* t_i *for each* $i, 1 \leq i \leq n$*. For an atomic formula* A *and a substitution* Σ*,* $A\Sigma$ *denotes applying* Σ *to* A *in the usual manner. For a conjunction* $C = A_1 \wedge \cdots \wedge A_n$*,* $C\Sigma$ *denotes the set* $\{A_1\Sigma, \ldots, A_n\Sigma\}$*.*

Definition 3.2 (Instances of Formulas)**.** *A (ground)* formula instance *is a pair* (ϕ, Σ) *of a formula* ϕ *and a (ground) substitution* Σ *of the universally quantified variables occurring in* ϕ*. We call a formula instance* (ϕ, Σ)*, where* $\phi = \forall \vec{X}\ [C \longrightarrow \exists \vec{Y}(C_1 \vee \cdots \vee C_m)]$*,* applicable *for a set of facts* F*, if* $C\Sigma \subseteq F$*, and for all substitutions* Σ' *with domain* $\vec{Y}$*, and for all* $i \in \{1, \ldots, m\}$*:* $(C_i\Sigma)\Sigma' \not\subseteq F$*.*

We will only be concerned with ground substitutions, and ground formula instances, hence "ground" will be omitted.

The proof search is defined below, but we give first a more informal explanation: It maintains a set of facts, which initially consists of the right-hand sides of all rules with empty left-hand sides. At each *step* in the proof search, the prover chooses an instance of a rule such that the left-hand side is in the fact-set, while the right-hand side is not completely covered by the fact-set. If the right-hand side is a conjunction, this is added to the fact-set and the prover continues. Otherwise, if there are several disjuncts in the right-hand side, the prover must treat each of these separately. The prover ends a branch if there is no applicable rule instance, or if it can infer a contradiction. The latter case includes the cases when there is an applicable rule instance with empty right-hand side. When there is no applicable rule instance, the current fact-set is immediately returned as a model of the theory. In the other case, a proof of contradiction of the current branch can be built.

Definition 3.3 (Proof Search)**.** *[3, 4] The algorithm* $A(F, T)$ *takes as input a set of facts* F*, and a coherent theory* T*. The output of the algorithm is either a list of formula instances or a set of facts.*

If there is no formula instance applicable in F*, the algorithm returns* F*. If there is an applicable instance of a formula with empty consequent, the output of the algorithm is that instance. Otherwise, an applicable instance* (ϕ, Σ)*, where* $\phi = \forall \vec{X}.(A_1 \wedge \cdots \wedge A_n \Rightarrow \exists \vec{Y}.(C_1 \vee \cdots \vee C_j))$*, is chosen. Assume* $\vec{Y} = Y_1, \ldots, Y_k$ *are the existentially quantified variables. Let* $c_1, \ldots, c_k$ *be fresh constants, and* $\Sigma' = \Sigma \cup \{Y_1 \leftarrow c_1, \ldots, Y_k \leftarrow c_k\}$*. Now, for each* $i, 1 \leq i \leq j$*, run recursively the algorithm* $A(F \cup (C_i\Sigma'), T)$*. If any of these runs return a set of facts, return one of these sets, otherwise, concatenate the lists of formula instances returned, prefix the list with* (ϕ, Σ)*, and return this list.*

If formula-instances are chosen in a fair manner, the algorithm is complete [3]. That is, $A(\{\}, T)$ will either return a set of facts which is a model of T, or it will return a list of formula instances representing a proof that the theory has no finite model.

The above proof procedure is intentionally underspecified: When several rules have applicable instances, it does not define which rule to choose. We will call a system of choosing between rules with applicable instances for a *strategy*. In addition, the algorithm for finding applicable rule instances is not specified. The latter is what we will call *matching*. Both matching and strategy are crucial in terms of increasing a prover's efficiency. Lastly, we do not specify which instance to choose when a rule has more than one applicable instance, except that this must be done in a fair manner.

In principle, improving the matching is orthogonal to improving the strategy. It should be noted that different algorithms for matching may list the applicable instances of one rule in different orders. Therefore, changing the matching while keeping the strategy fixed, may change the steps taken when proving a theorem. The rules containing applicable instances must be the same for all matchers. We have not seen any interesting strategy that differentiates between different instances of the same formula, so this difference does not really interfere with the proper functioning of the strategy.

Note further that the prover spends almost all its time doing matching. The remainder of the provers functionality takes comparatively little time, ca. 1%. A common implementation of the matching is to do search the fact-set for matching facts for the left-hand side. This process loses information about partially fulfilled left-hand sides. The partial satisfaction must be redone on every step until a full satisfaction is found. A solution to this problem could be to add lemmas corresponding to the partially instantiated rules. But this is very costly in terms of space and time.

Our goal in this paper is to investigate the usage of the RETE algorithm for the matching. We will see that it eliminates the redoing of partially satisfied rules, while still not storing the whole lemmas as suggested above. The intuition is that the fact-set is inserted into the RETE network, which will output a list of rule instances where the left-hand side is satisfied by the fact-set. The proof search will then use this list to look for applicable instances.

4 The RETE Algorithm

RETE is an algorithm for the "Many Pattern/Many Object Pattern Match Problem" initially developed for *production systems* [6]. As shown in section 2 a production system consists of a fixed set of *productions*, and a *working memory*. In the setting of coherent logic, the productions are the axioms and the working memory is the fact-set. If there are elements in the working memory matching consistently all patterns in the left-hand side of a production rule, the actions in the right-hand side are executed. The types of actions is not closely specified in RETE, but a consequent in coherent logic can be seen as actions of certain types. Hence, an axiom in coherent logic can be seen as a production rule in a production system. This correspondence spurred use of the RETE algorithm for coherent logic.

The RETE algorithm can be used to find applicable rule instances, given a set of production rules and a fact-set. More specifically, we will use RETE to find the rule instances $(\forall \vec{X}\ [C \longrightarrow \exists \vec{Y}(C_1 \vee \cdots \vee C_n)], \Sigma)$ such that $C\Sigma \subseteq F$. Recall from Definition 3.3 that this is necessary, but not sufficient for a rule instance to be applicable. The second condition for applicability, namely that for all Σ' with domain $\vec{Y}$ and all $i \in \{1, \ldots, n\}$, $C_i\Sigma\Sigma' \not\subseteq F$, will not be checked using RETE, although it is possible. The prover filters out these instances where elements contained inside the fact-set can take the place of newly generated substitutions using a straight-forward search of the fact-set. (Note that the issue with redoing partial matches is not an issue here, since each rule instance is only tested once. Hence, there is no need to use a RETE net for this.)

Name	Values	Def. on
`type`	α, β, or rule	all
`formula`	atomic or coherent formula	α, rule
`children`	list of links β	α
`child`	link to β | *rule*-node	β
`leftParent`	*null* or link to a β	β
`rightParent`	link to a α	β
`store`	substitutions	all
`freeVars`	list of variables	β, rule

Table 1: Fields of the nodes in a RETE network

The RETE algorithm consists of two steps, the first step is executed only once, while the second is executed repeatedly. The first step, explained in Section 4.1, is to create the RETE-network from the production rules. The second step, explained in Section 4.2 takes an element from the working memory, *inserts* it into the RETE-network, and outputs 0 or more new instances of productions that are now satisfied.

The main difference between a general production system, and our proof system is that disjunctions in the conclusion creates a *fork*, where each disjunct becomes its own branch, with its own working memory.

4.1 Constructing the RETE-Network

The RETE-network consists of three types of nodes: α-nodes, β-nodes and *rule*-nodes. (In [6] α- and β-nodes are called *one-input* and *two-input* nodes, respectively.) *Links* between nodes are indicated by arrows, i.e. $a \to b$ reads a has a pointer to b, they can be bidirectional; $a \leftrightarrow b$.

4.1.1 The Rule-Specific Nodes

For each axiom, there is a corresponding *rule* node, which will construct the final output of the RETE-network in the second step of the algorithm. There is one β-node corresponding to each atom/conjunct in the left-hand side of each rule. There is one α-node for each atomic formula occurring on the left-hand side of any rule. Since the same atom may occur in several precedents, each α-node may correspond to several β-nodes. Each α-node has one or more links to the corresponding β-node(s). The β-nodes that represent the left hand side of a rule are linked in a string, starting with a *dummy*-node, and ending with the last β-node being linked to a *rule*-node. Each α-node has a store containing substitutions of the free variables occurring in the atom that the α-node represents. Each β-node except the *dummy*-node has "input arrows" from the α-node, and from its preceding β-node, and an "output arrow". The output arrow connects to the β-node representing the atom to its right, except for the rightmost β-node which has an arrow to the *rule*-node.

We will now give a more formal description of the algorithm, after defining the necessary data types.

Definition 4.1. *A node in a RETE network is a data structure with fields as described in Table 1. All fields are filled with values in part* 1 *of the algorithm and kept unchanged in part* 2*, with exception of the* store *field. The* store *field is empty after part* 1 *and filled with values in part* 2 *of the RETE algorithm.*

Example 4.2. *In Fig. 4 the partial RETE-network constructed from a generic rule / axiom is presented. Each requirement (atomic formulas inside the premise in CL) gets its own α-node, these can be shared between β-nodes as with axiom 4 and 5 from Fig 1. The β-nodes are linked together with* `leftParent` *and* `child` *links; following the links from left to right gives us the requirements of the axiom / rule. The leftmost* `leftParent` *is a null reference (dummy-node), illustrated by a diamond ($\diamond$) in the figures. The dummy-node indicates that we have reached the first β-node. Note the* `formula` *field inside the α-nodes and the rule-node, representing the requirements inside the α-nodes and the entire rule / axiom inside the rule-node.*

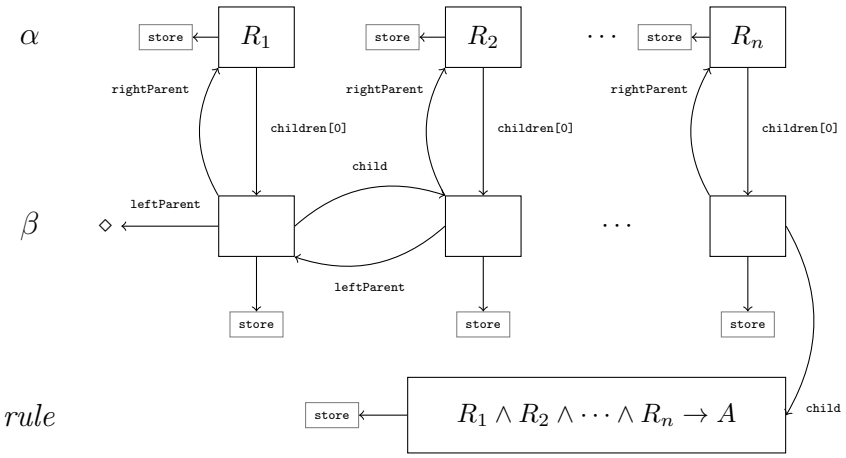

Figure 4: From axioms / rules to RETE-networks

Example 4.3. *Fig. 5 shows the RETE-network constructed for the theory shown in Fig. 1.*

4.1.2 Representing the RETE-network

As can be seen in figure 5 α-nodes can be shared by different axioms / rules, when their requirements are syntactically identical, but this can be extended to all α-nodes that can be unified through renaming ($p(X, Y)$ and $p(Y, X)$ for instance), since these terms will match the same facts. Shared α-nodes of syntactically different nodes introduce more complex bookkeeping, since the prover must keep track of the mappings between the different substitutions. Therefore we left this sharing out of the implementation. The numbers inside the *rule*-nodes correspond to the axioms given in Fig. 1, i.e., the first axiom/rule which introduces the fact-set ($\top \longrightarrow p(a)$) is not part of the network. Each of the α-nodes holds a list of matching substitutions, this is left out of Fig. 5 to simplify the presentation. See Fig. 6 for a more detailed look at the substitution lists.

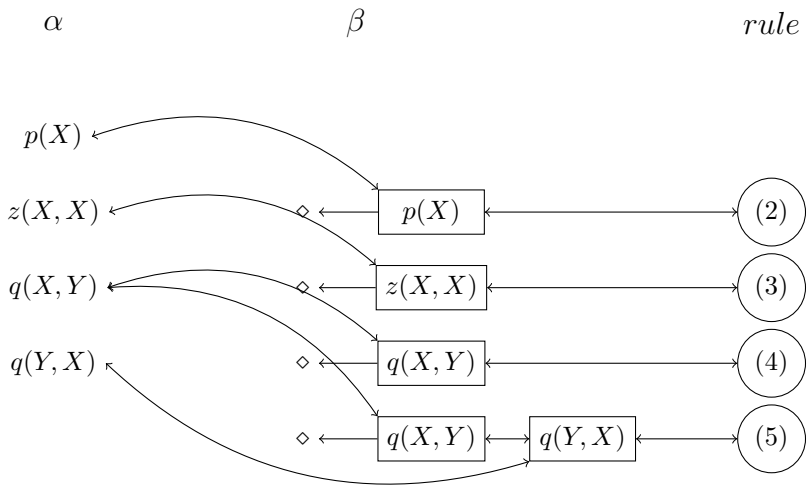

Figure 5: Simplified RETE-network for the theory in Fig. 1

4.2 Inserting Facts into the Network

When facts are inserted into the RETE-network, the first step is to find the matching substitutions between the facts of the fact-set and the atoms in the α-nodes. The fact is "inserted" into each α-node, in the following way: The α-node tries unify the *inserted* fact and the atom it represents. If no unifier is found, the α-node can safely ignore this fact. Otherwise, the unifying substitution is added to the matching α-node's `store`, and the β-nodes connected to the α-node are notified. The β node will check whether any of the stored substitutions do not have conflicting variable assignments with the inserted substitution. If that is the case, the union will be passed on to the next β-node. The next β-node will check that it has not seen the substitution before, and then tries to take the union with all the substitutions in the store of the corresponding α-node. Any successful unions are passed on to the next β-node. This process is repeated at the next node until no union is found, or till it reaches the rule node. The rule node outputs the substitution as a rule instance on the queue.

Unification of terms and of sets of terms is done in the usual way, as defined in e.g. Martelli & Montanari [10]. unify$(,)$ returns a most general unifier or "false". In Listings 1 and 2 we show the basic algorithm for inserting a fact into a RETE network.

5 Optimizing the Algorithm

The algorithm shown in Sect. 4 is a rather direct application of RETE. In this section we will explore and motivate some optimizations and modifications.

5.1 Computational Complexity

The decision problem we are trying to solve falls into the category of undecidable problems. That gives us little information about the complexity of the decidable fragment, which all theorems fall into. The problem is computationally hard, and the complexity of the proof

Listing 1: Insert facts into RETE network

```
foreach α−node a in the network
  s = unify(a.formula, f);
  if s ≠ false;
    Replace variable−names in f with fresh names;
    foreach t ∈ a.store
      v = unify(σ, t);
      if v ≠ false;
        Exit algorithm;
      put s on a.store;
      foreach c ∈ a.children do
        if c.leftParent == null
          run Listing 2 on c.child, s;
        else
          foreach u ∈ c.store
            v = unify(u, s);
            if v ≠ false
              run Listing 2 on c.child, v;
```

Listing 2: Insert substitution s into β- or rule-node n

```
if s ∉ n.store
  put s on n.store;
  if n.type ==rule
    push (n.formula, s) on queue;
  else
    foreach s′ ∈ n.rightParent.store
      u = unify(s, s′);
      if u ≠ false
        run algorithm recursively on n.child, u|_{n.freeVars};
```

search stems from the exponential explosion of different ways to *match* requirements in the left hand side of (production) rules/axioms. For a rule with n requirements:

$$R_1 \wedge \ldots \wedge R_n \longrightarrow A \tag{13}$$

and m ways to match each requirement, we get m^n possible ways to fulfill this rule's left hand side. Many of these may lead to variable conflicts and cannot be applied, but the prover still needs to figure that out.

The exponential number of possible matches is a problem both in terms of time complexity and storage, and we need an algorithm for a lazy evaluation.

5.2 Matching

In 5.1 we saw that the number of possible matches for a rule is basically the cross-product of all the requirement's individual matches, so an eager evaluation of the possible matches is not always practical.

By storing a list of each predicates' matching substitutions, we get a structure similar to the one displayed in Fig. 6.

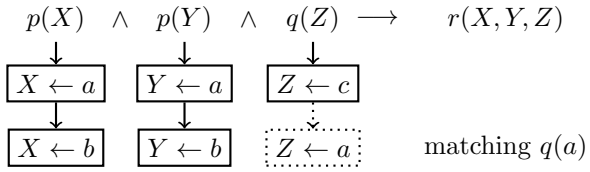

Figure 6: Matching substitutions

Assume that $q(a)$ is a new fact that we have matched against the left hand side of this rule, and we would like to know whether or not this produced a match, and if so which matching substitutions did it generate. The result is the cross product of the substitutions gathered by matching facts against $p(X)$ and $p(Y)$ with the new substitution generated by matching $q(a)$ against the last predicate $q(Z)$, i.e. $\{Z \leftarrow a\}$. In this example there is no variable conflict, so every substitution is valid. The set of substitutions obtained by matching the fact $q(a)$ against the axiom/rule can be seen in Fig. 7.

$$\{X \leftarrow a, Y \leftarrow a, Z \leftarrow a\} \tag{14}$$
$$\{X \leftarrow a, Y \leftarrow b, Z \leftarrow a\} \tag{15}$$
$$\{X \leftarrow b, Y \leftarrow a, Z \leftarrow a\} \tag{16}$$
$$\{X \leftarrow b, Y \leftarrow b, Z \leftarrow a\} \tag{17}$$

Figure 7: Matching substitutions

5.3 Laziness

The formulation of RETE we have given above outputs *all* new applicable rule instances. In comparison, the standard approach for matching will only need to run until one instance is found. Since we do not apply any strategies which make use of more than one instance of each axiom, it is unnecessary to find more than one instance of each axiom at any step. In addition, for some theories, so many instances may be generated of some rules, that the prover cannot generate all in the time available. It is therefore desirable to let the prover use the first rule instance that is generated, rather than making it wait until all instances of each rule have been output by the RETE network. This could be achieved by pausing or halting the RETE network when it outputs a rule instance, and store the full state of the network. Another option is to use a multithreaded/parallel solution; In addition to the main thread for the prover, we run

one separate thread for each axiom. Note that large parts of the RETE network, including all beta-nodes, and most alpha-nodes, are specific for certain axioms. Each axiom-specific thread takes care of the RETE algorithm from the point it enters a node specific for the corresponding axiom. Instances are generated by the thread, and pushed to the queue of rule instances, which the prover main-thread can then access. This approach eliminates the need for generating all applicable rule instances, and in addition adds some parallelism to the prover.

5.4 Sharing Between Branches

Recall that the prover must split into branches when it is treating an instance of an axiom with a disjunction on the right-hand side. The whole RETE-network, except the `store`s, can be shared or reused between these branches, as the content does not change. The stores (lists of substitutions) cannot be completely shared, as substitutions added in one branch, might not be added in other branches. In a single-threaded prover this can be handled by removing substitutions when backtracking. This would be analogous to the removal of facts from the fact-set done during backtracking by other provers, e.g. CL.pl [3]. In a multi-threaded implementation this approach is not applicable. However, the common parts can be shared by implementing the stores with linked lists. Each branch must then maintain, for each α- and β-node, a separate link to the *topmost* substitution in each store. Each substitution contains a link to the next substitution, or a null-pointer. When branching, the links to the topmost substitutions are copied. This way, a high degree of sharing is achieved.

The queue of rule instances output by the RETE network cannot be shared in a similar manner in multi-threaded implementations. In a single-threaded implementation it is possible to avoid copying the queue by taking the following steps: Maintain a separate queue for each axiom, instead of a single queue with all instances. Note that in these queues, all elements are inserted in one end, and deleted in the other end. We implement these with arrays, where elements are not deleted, but instead, a pointer to the current location of each end is updated. At branching points in the proof, we store the locations of the two ends of each queue. When backtracking, we only need to use the stored locations of the ends of the queues.

6 Results

We have implemented a multithreaded version of the RETE algorithm, as described above, in a prover for coherent logic[1]. The prover takes coherent theories as input, and uses the algorithm described to search for a proof of inconsistency. If it finds such a proof, it can be output in a form readable by the Coq proof assistant. Otherwise, if the prover finds a model of the theory, this can also be output.

We have compared the implementation with three exisiting provers for coherent logic, Geo [4], CL.pl [3], and colog [5], and for reference, with three standard provers for first-order logic: E[2], Vampire[3], and leanCop[4]. The test set was provided by M. Bezem, and contains in all 64 tests[5]. All the test formulas are written in the format of coherent logic, without functions and equality, and preformed using an Intel© Core™ i5-750 2.67GHz processor. We let each prover run up to 60 seconds on each test; in Table 3 the time used by the different provers to solve the

[1] Available at `http://code.google.com/p/clp`
[2] Version 1.4 Namring, downloaded from `http://www4.informatik.tu-muenchen.de/~schulz/E/E.html`
[3] Version 0.6 (revision 903), downloaded from `http://www.vprover.org`
[4] Version 2.1, downloaded from `http://leancop.de`.
[5] Available at `http://code.google.com/p/clp/source/browse/?repo=test`.

problems are given, a `timeout` indicates that no solution was produced within 1 minute. Note that formulas from the test-set were all provers succeed within a short time-frame are removed from Table 3, the complete coverage of the test-set can be seen in Table 2. The intention is obviously not to rank the provers; but to illustrate that the problems in the test-set are hard even for *state of the art* provers based on different calculi (connection calculus and resolution). Furthermore, we wish to show that using RETE for the matching (clp) can contribute to the speed of coherent provers on these tests.

Table 2: Comparison

	Geo	CL.pl	clp	Colog	vampire	Eprover	leanCop
Successes	58	46	57	57	55	54	34

7 Related Work and Conclusion

7.1 Related Work

The RETE algorithm was introduced for *production systems* in artificial intelligence by Forgy [6]. The RETE algorithm has been applied in automated reasoning earlier, specifically with hyper-linking, in Lee & Wu [9]. The original production systems would usually consist of a large amount of rules, and saving space in the network is of importance. The number of axioms in a typical theorem is much smaller, so saving the number of nodes in the network is not so important here. On the other hand, theorem provers may generate large amunts of facts, many

Table 3: Runtime and results for each prover on some of the tests. The fastest time for each test is emphasized.

Problem	Geo	CL.pl	clp	Colog	vampire	Eprover	leanCop
anl	0.653s	timeout	*0.014s*	0.153s	12.240s	2.677s	timeout
cdp	0.106s	timeout	*0.007s*	0.126s	0.013s	0.011s	timeout
cro_8_2	timeout	0.262s	2.355s	*0.373s*	12.883s	21.642s	timeout
five	0.283s	timeout	*0.092s*	0.287s	timeout	timeout	timeout
latt	timeout	*0.170s*	0.174s	1.634s	timeout	timeout	timeout
len	0.676s	timeout	timeout	*0.205s*	timeout	timeout	timeout
mb	0.064s	0.031s	0.016s	0.159s	*0.007s*	0.009s	1.019s
nl	0.891s	0.020s	*0.014s*	0.194s	12.233s	14.823s	timeout
nnl	0.347s	47.661s	*0.053s*	0.253s	12.238s	15.295s	timeout
p1p2	timeout	timeout	2.515s	*1.489s*	timeout	timeout	timeout
p2p1	timeout	*0.105s*	0.446s	0.476s	timeout	30.995s	timeout
pp	timeout	timeout	*2.237s*	3.239s	timeout	timeout	timeout
qedf	0.007s	timeout	*0.005s*	0.222s	0.006s	0.008s	1.019s
sd	43.444s	timeout	*0.421s*	timeout	45.691s	timeout	timeout
timeouts	5	8	1	1	6	6	12
fastest	0	2	8	3	1	0	0

more than it can treat in the time given. This lead to the "laziness" approach for our matcher, which is not seen in [6] or [9]. The disjunctions and existential quantifiers in the right-hand side of the rules / axioms require modifications to the RETE algorithm which have not previously been investigated.

7.2 Conclusion

We have shown how the RETE algorithm can be modified for use in a forward chaining procedure. We also describe some optimizations, namely lazyness and sharing of data between threads. proposed.

Acknowledgements

We would like to thank Roger Antonsen, Marc Bezem, and Andrew Polonsky for discussing these topics with us.

References

[1] Peter Baumgartner, Ulrich Furbach, and Ilkka Niemelä. Hyper tableaux. In José Júlio Alferes, Luís Moniz Pereira, and Ewa Orlowska, editors, *JELIA*, volume 1126 of *Lecture Notes in Computer Science*, pages 1–17. Springer, 1996.

[2] Marc Bezem and Thierry Coquand. Newman's lemma - a case study in proof automation and geometric logic, logic in computer science column. *Bulletin of the EATCS*, 79:86–100, 2003.

[3] Marc Bezem and Thierry Coquand. Automating coherent logic. In Geoff Sutcliffe and Andrei Voronkov, editors, *LPAR*, volume 3835 of *Lecture Notes in Computer Science*, pages 246–260. Springer, 2005.

[4] Hans de Nivelle and Jia Meng. Geometric resolution: A proof procedure based on finite model search. In Ulrich Furbach and Natarajan Shankar, editors, *IJCAR*, volume 4130 of *Lecture Notes in Computer Science*, pages 303–317. Springer, 2006.

[5] John Fisher. Colog program. `http://johnrfisher.net/colog/index.html`, 2011.

[6] Charles Forgy. Rete: A fast algorithm for the many patterns/many objects match problem. *Artif. Intell.*, 19(0):17–37, 1982.

[7] Ryuzo Hasegawa, Hiroshi Fujita, and Miyuki Koshimura. Mgtp: A model generation theorem prover - its advanced features and applications. In Didier Galmiche, editor, *TABLEAUX*, volume 1227 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 1997.

[8] Shie-Jue Lee and David A. Plaisted. Eliminating duplication with the hyper-linking strategy. *J. Autom. Reasoning*, 9(1):25–42, 1992.

[9] Shie-Jue Lee and Chih-Hung Wu. Improving the efficiency of a hyperlinking-based theorem prover by incremental evaluation with network structures. *J. Autom. Reasoning*, 12(3):359–388, 1994.

[10] Alberto Martelli and Ugo Montanari. An efficient unification algorithm. *ACM Trans. Program. Lang. Syst.*, 4(2):258–282, 1982.

[11] Chih-Hung Wu and Shie-Jue Lee. Parallelization of a hyper-linking-based theorem prover. *J. Autom. Reasoning*, 26(1):67–106, 2001.