

Efficient Rule-Matching for Automated Coherent Logic

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

Abstract

Coherent logic is a syntactically defined fragment of first-order logic. The paper describes an experiment with a *prover* for coherent logic. A prover here means software that takes as input a theory in coherent logic, and (if it halts) either outputs a proof of a contradiction in the theory, or a model of the theory.

Existing provers for coherent logic typically spend much of their time in the process of matching inferred 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 coherent logic 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 proof search in coherent logic but not in production systems.

1 Introduction

The goal of this paper is to illustrate how the *Rete*¹ algorithm [6] can be modified to fit the matching problem of a proof procedure for coherent logic. *Coherent Logic* (CL) [12, 3] or *Geometric Logic* [4] is a fragment of first-order logic.²The proof procedure matches an expanding set of atoms against a static set of negative clause literals and 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.

Although we exploit similarities between production systems and coherent logic, there are important differences: Coherent logic has branching and usually many more facts are inferred than in production systems. In general we cannot generate all consequences since there are too many. The adaption and refinement of Rete to handle these issues is the main contribution of the paper. We also include an evaluation of the usefulness of Rete in this context by comparing with other provers.

This paper was presented at the NIK-2013 conference; see <http://www.nik.no/>.

¹“Rete” is not an acronym, rather, it is inspired by the Italian word for network.

²Coherent Logic is defined as the first-order fragment of Geometric Logic. Geometric logic is in topology the fragment of higher-order logic describing toposes. Some authors also refer to the first-order fragment as Geometric Logic. For an overview, see Chapter 1 of [10].

Coherent Logic

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

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

Formulas of such form are called *coherent formulas* or *axioms*. 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 $\text{vars}(\Phi)$. 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 exist 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 \perp (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 at least 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.

$$\begin{aligned} \top &\longrightarrow p(a) & (1) \\ p(X) &\longrightarrow z(X, X) & (2) \\ z(X, X) &\longrightarrow \exists Y, Z : q(X, Y) \vee q(X, Z) & (3) \\ q(X, Y) &\longrightarrow q(Y, X) & (4) \\ q(X, Y) \wedge q(Y, X) &\longrightarrow \perp & (5) \end{aligned}$$

Figure 1: A coherent theory

The Rete algorithm has been successfully used for other types of reasoning in the past, in [8, 13] they optimize the process of hyper-linking [7]. 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 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).

$$R_1 \wedge R_2 \wedge \dots \wedge R_n \rightarrow A \quad (6)$$

The intuition is that fulfilling the requirements (R_1, \dots, 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.

$$p, q \Rightarrow r \quad (7)$$

$$r, p \Rightarrow t \quad (8)$$

$$t, q \Rightarrow goal \quad (9)$$

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.

$$\mathbf{p}, \mathbf{q} \Rightarrow r \quad (10)$$

$$r, \mathbf{p} \Rightarrow t \quad (11)$$

$$t, \mathbf{q} \Rightarrow goal \quad (12)$$

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 performed, resulting in the new element: t . The term t fulfills the last requirement of the third production-rule: $\mathbf{t}, \mathbf{q} \Rightarrow goal$. 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* imply that we only have to process the working memory elements once, i.e., the rules will always match the same working memory 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].

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.

Matching in this context refers to a regular first order unification [9], 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.

At this point it should be emphasized that Rete will only be used for the matching part in the prover. This implies that the choice the prover takes at each step (which rule instance to apply) is independent of the use of Rete.

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 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. $\{X_1 \leftarrow t_1, \dots, X_n \leftarrow t_n\}$ 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 $C = A_1 \wedge \dots \wedge A_n$, $C\Sigma$ denotes the set $\{A_1\Sigma, \dots, A_n\Sigma\}$.

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 $\phi = \forall \vec{X} [C \rightarrow \exists \vec{Y} (C_1 \vee \dots \vee C_m)]$, applicable for a set of facts F , if $C\Sigma \subseteq F$, and for all substitutions Σ' with domain \vec{Y} , and for all $i \in \{1, \dots, m\}$: $(C_i\Sigma)\Sigma' \not\subseteq F$.

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

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 $\phi = \forall \vec{X}. (A_1 \wedge \dots \wedge A_n \Rightarrow \exists \vec{Y}. (C_1 \vee \dots \vee C_j))$, is chosen. Assume $\vec{Y} = Y_1, \dots, Y_k$ are the existentially quantified variables. Let c_1, \dots, c_k be fresh constants, and $\Sigma' = \Sigma \cup \{Y_1 \leftarrow c_1, \dots, Y_k \leftarrow c_k\}$. 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, if it terminates, 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 latter proof is in the style of natural deduction and will be a tree. The root is an instance of a formula with empty left-hand side. The branching nodes are instances of formulas with disjunction on the right-hand side, and the leaves are instances of formulas with empty right-hand sides. The remaining internal nodes have only one disjunct in the right-hand side. The formula instance at each node has only conjuncts on the left-hand side which also appear in the right-hand side on nodes above it in the tree.

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 a *strategy*. (In our implementation we only use deterministic strategies.) 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; the choice of which rule instance to apply should not depend on the algorithm used to find applicable rule instances. It should though be noted that different algorithms for matching may list the applicable instances of the same 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 of course be the same for all correct 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%.³ A common implementation of the matching is to 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.

³We measured this on our own implementation (see Section 6) using gprof.

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 are 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 $(\forall \vec{X} [C \rightarrow \exists \vec{Y} (C_1 \vee \dots \vee C_n)], \Sigma)$ 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 is that the right hand side is not true in the fact-set. Or, formally, that for all Σ' with domain \vec{Y} and all $i \in \{1, \dots, n\}$, $C_i \Sigma' \not\subseteq F$. The prover filters out these instances using a straight-forward search of the fact-set.

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, is to create the Rete-network from the production rules. The second step, explained in Section 4 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 create a *fork*, where each disjunct becomes its own branch, with its own working memory.

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. $a \rightarrow b$ reads a has a pointer to b , they can be bidirectional; $a \leftrightarrow b$.

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 α -node for each atomic formula occurring on the left-hand side of any rule. This node will store substitutions of the corresponding atomic formula. There is one β -node corresponding to each atom/conjunct in the left-hand side of each rule. A β -node will contain substitutions of the part of the left-hand side to the left of the corresponding conjunct. 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

Table 1: Fields of the nodes in a Rete network

Name	Values	Defined for nodes of type
type	α , β , or rule	all types
formula	atomic or coherent formula	α and rule
children	list of links to a β	α
child	link to a β <i>rule</i> -node	β
betaParent	<i>null</i> or link to a β	β
alphaParent	link to an α	β
store	substitutions	all types
freeVars	list of variables	β and rule

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 atomic formula inside the premise 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 *betaParent* and *child* links; following the links from left to right we pass through nodes representing all atoms in the premise of the axiom / rule. The leftmost *betaParent* 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. The *formula* field of the α -node links to the corresponding atom, while the *rule*-node has a link to the entire formula.*

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

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.

Inserting Facts into the Network

Recall that α nodes store substitutions of the corresponding atomic formulas. On the other hand β nodes store substitutions of the part of the premise to the left of the corresponding

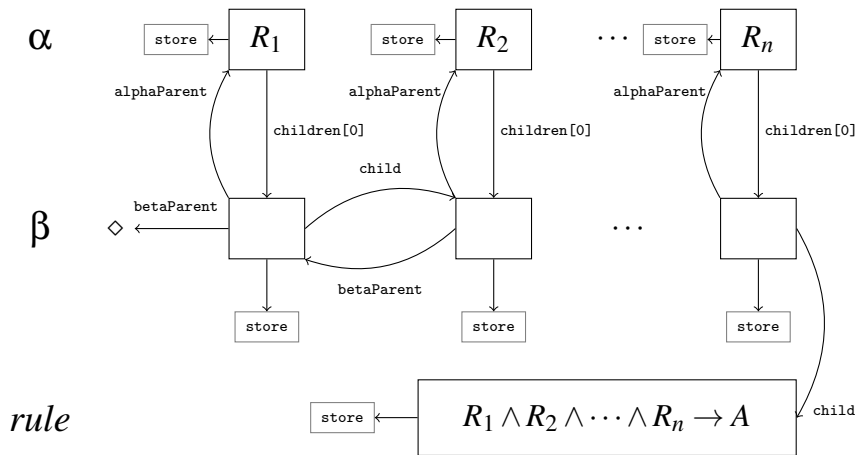


Figure 4: From axioms / rules to Rete-networks

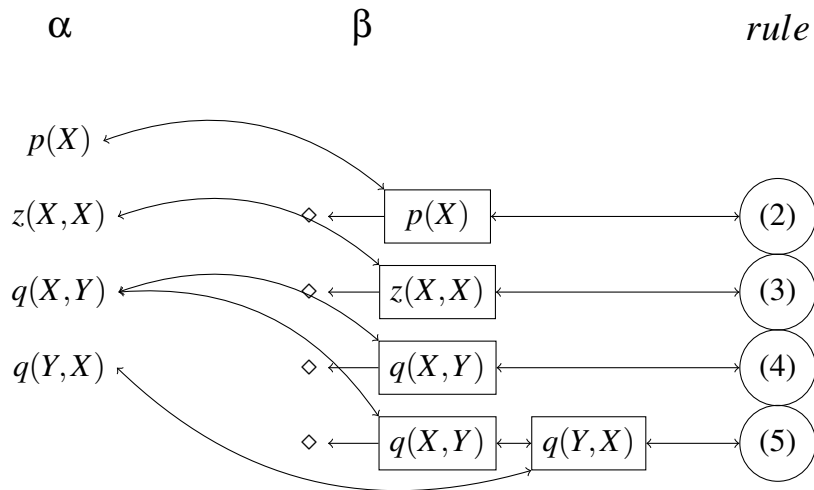


Figure 5: Simplified Rete-network for the theory in Fig. 1

atomic formula. 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 to 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 [9]. `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.

Listing 1: Inserting a fact f into a Rete network

```

foreach  $\alpha$ -node  $a$  in the network
   $s = \text{unify}(a.\text{formula}, f)$ ;
  if  $s \neq \text{false}$ ;
    foreach  $t \in a.\text{store}$ 
      if  $t = s$ 
        Exit algorithm;
      put  $s$  on  $a.\text{store}$ ;
    foreach  $c \in a.\text{children}$  do
      if  $c.\text{betaParent} == \text{null}$ 
        run Listing 2 on  $c.\text{child}, s$ ;
      else
        foreach  $u \in c.\text{store}$ 
          if ( not conflicting_variable_assignments( $u, s$ ) )
            run Listing 2 on  $c.\text{child}, u \cup s$ ;

```

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

```

if  $s \notin n.\text{store}$ 
  put  $s$  on  $n.\text{store}$ ;
  if  $n.\text{type} == \text{rule}$ 
    push ( $n.\text{formula}, s$ ) on queue;
  else if  $n.\text{type} == \beta$ 
    foreach  $s' \in n.\text{alphaParent}.\text{store}$ 
      if ( not conflicting_variable_assignments( $s, s'$ ) )
        run algorithm recursively on  $n.\text{child}, u|_{n.\text{freeVars}}$ ;

```

5 Laziness

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.

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 multi-threaded/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.

6 Results

We have implemented a version of the Rete algorithm with separate threads for each axiom, as described in Section 5, in a prover for coherent logic called *clp*.⁴ 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 our implementation (*clp*) with three existing provers for coherent logic, *Geo* [4], *CL.pl* [3], and *colog* [5], and for reference, with three standard provers for first-order logic: *E*⁵, *Vampire*⁶, and *leanCop*⁷. The test set was provided by M. Bezem, and contains in all 66 tests⁸. All the test formulas are written in the format of coherent logic, without functions and equality, and performed using an Intel© Core™ i5-750 2.67GHz processor (with 4 processor cores). We let each prover run up to 60 seconds on each test. The complete coverage of the test-set can be seen in Table 2: *successes* indicates the number of tests on which the prover produced a correct answer, while *fastest* is the number of times a prover gave an answer in shorter time than the other provers. Table 3 shows more details on some of the test. 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 (as in our implementation *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	47	59	57	55	53	34
Fastest	14	6	24	2	14	5	0

7 Related Work and Conclusion

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 hyperlinking, in Lee & Wu [8]. The original production systems

⁴Available at <http://code.google.com/p/clp>

⁵Version 1.6 Tiger Hill, downloaded from <http://www4.informatik.tu-muenchen.de/~schulz/E/E.html>

⁶Version 0.6 (revision 903), downloaded from <http://www.vprover.org>

⁷Version 2.1, downloaded from <http://leancop.de>.

⁸Available at <http://code.google.com/p/clp/source/browse/?repo=test>.

Table 3: Runtime and results for each prover on some of the tests. The fastest time for each test is emphasized. *timeout* means that no solution was produced in 60 seconds. *memory* means that all available memory was exhausted.

Problem	Geo	CL.pl	clp	colog	vampire	eprover	leancop
anl	0.639s	timeout	<i>0.03s</i>	0.141s	12.247s	0.519s	timeout
cdp	0.109s	timeout	<i>0.01s</i>	0.108s	0.019s	0.015s	timeout
cro_8_2	timeout	<i>0.251s</i>	0.948s	0.337s	12.87s	timeout	timeout
five	0.283s	<i>0.105s</i>	0.142s	0.231s	timeout	timeout	timeout
latt	timeout	<i>0.17s</i>	0.293s	0.956s	timeout	timeout	timeout
len	0.677s	timeout	memory	<i>0.129s</i>	timeout	timeout	timeout
mb	0.069s	0.048s	0.025s	0.112s	<i>0.007s</i>	0.018s	1.034s
nl	0.862s	<i>0.017s</i>	0.025s	0.133s	12.238s	0.221s	timeout
nnl	0.343s	0.227s	<i>0.147s</i>	0.172s	12.235s	0.26s	timeout
p1p2	timeout	0.916s	1.05s	<i>0.881s</i>	timeout	timeout	timeout
p2p1	timeout	<i>0.102s</i>	0.299s	0.31s	timeout	timeout	timeout
pp	timeout	40.035s	<i>0.983s</i>	1.92s	timeout	timeout	timeout
qedf	0.009s	timeout	0.011s	0.112s	<i>0.004s</i>	0.007s	1.035s
sd	43.254s	timeout	<i>0.674s</i>	timeout	45.266s	timeout	timeout
tdpe2	1.738s	timeout	timeout	timeout	24.958s	<i>0.46s</i>	timeout
tdpe3	3.278s	38.007s	timeout	timeout	24.939s	<i>0.854s</i>	timeout
tdpe4	1.002s	<i>0.036s</i>	timeout	timeout	13.051s	0.414s	timeout

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 theory 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 amounts of facts, many more than it can treat in the time given. This led to the “laziness” approach for our matcher, which is not seen in [6] or [8]. 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.

Identical atoms inside the premises of a coherent theory will collapse onto a single α -node, which can be seen as a very primitive term indexing [11] scheme. For deep terms this would not be very useful, but the coherent theories we have worked with so far have contained mostly shallow terms. As described in [1], term indexing does not necessarily have positive effects when operating on shallow terms. It should be noted however that the Rete algorithm does not stand in the way of building a term index to improve matching, and it is certainly something that should be investigated as a future improvement.

Conclusion

We have shown how the Rete algorithm can be modified for use in the *matching* in a forward chaining procedure. This is non-trivial, as the proof procedures we consider have branching and in general produce too many facts that we can consider all. We also describe some refinements of Rete, namely laziness and sharing of data between threads. Finally, we have evaluated the usefulness of the approach by comparing our implementation of coherent logic and Rete with other provers.

Acknowledgments

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

References

- [1] Peter Baumgartner, Alexander Fuchs, and Cesare Tinelli. Implementing the model evolution calculus. *International Journal on Artificial Intelligence Tools*, 15(1):21–52, 2006.
- [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] Shie-Jue Lee and David A. Plaisted. Eliminating duplication with the hyper-linking strategy. *J. Autom. Reasoning*, 9(1):25–42, 1992.
- [8] 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.
- [9] Alberto Martelli and Ugo Montanari. An efficient unification algorithm. *ACM Trans. Program. Lang. Syst.*, 4(2):258–282, 1982.
- [10] Andrew Polonsky. *Proofs, Types, and Lambda Calculus*. PhD thesis, University of Bergen, 2010.
- [11] I. V. Ramakrishnan, R. C. Sekar, and Andrei Voronkov. Term indexing. In John Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, pages 1853–1964. Elsevier and MIT Press, 2001.
- [12] Thoralf Skolem. Logisch-kombinatorische untersuchungen ber die erfillbarkeit und beweisbarkeit mathematischer stze nebst einem theorema ber dichten mengen. *Skripter I*, 4:1–36, 1920.
- [13] Chih-Hung Wu and Shie-Jue Lee. Parallelization of a hyper-linking-based theorem prover. *J. Autom. Reasoning*, 26(1):67–106, 2001.