Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 238 (2009) 173–188
www.elsevier.com/locate/entcs

Proof Search for the First-Order Connection Calculus in Maude

Bjarne Holen, Einar Broch Johnsen and Arild Waaler [1]

Department of Informatics, University of Oslo, Norway

Abstract

This paper develops a rewriting logic specification of the connection method for first-order logic, implemented in Maude. The connection method is a goal-directed proof procedure that requires a careful control over clause copies. The specification separates the inference rule layer from the rule application layer, and implements the latter at Maude's meta-level. This allows us to develop and compare different strategies for proof search.

Keywords: First-order logic, connection method, rewriting logic, reflection, meta-programming, Maude

1 Introduction

The increasing use of logics in practical applications, and in particular non-classical logics, poses challenges for automated reasoning. A key issue currently addressed by the automated reasoning community, is how one can improve scalability of already successful methods. Besides optimizing the implementations, this can primarily be achieved in two ways: either by improving the proof calculus or by finding more clever ways of applying the rules.

This observation motivates the Maude implementation of the connection calculus presented in this paper. The implementation addresses first-order logic (FOL) without equality on clausal form. It is designed to satisfy two guiding principles. First, it clearly separates a rule layer from a strategy layer that governs rule application. Second, the set of rules that comprises the proof system is not restricted to FOL only, but has also non-classical counterparts. This way the implementation is part of a more wide-ranging project outlined in the discussion of future work in Section 7, in which we intend to contribute with both improved calculi and with

[1] Email: bjarneh@ifi.uio.no, einarj@ifi.uio.no, arild@ifi.uio.no

doi:10.1016/j.entcs.2009.05.019

more flexible strategies for rule application. The present work can be taken as a preliminary report from this activity.

Our focus on the strategy level makes Maude attractive as an implementation platform. This is partly due to its support for reflection [6], which allows strategic choices at run-time [5]. In particular, this gives a satisfying separation between deduction and strategic choices, allowing us to experiment with different strategies over the same deductive core. Ideally, theorem proving can be as simple as constructing a specification containing all the deductive rules (as rewrite rules), and then rewriting an appropriate term. This also gives a close relationship between the calculus and its specification.

The main rationale behind the isolation of a strategy layer is that more easily understood procedures are less error-prone than procedures in which rules and strategies are intermixed. Moreover if one opts for the other extreme, i.e., to build the strategy into the rules, the strategy will most likely be hard to modify. In practice one will often need to have a little of both, even if one tries to separate the layers, since pruning the search space may in some cases be easier to implement by means of inference rules than tacticals.

Our deductive platform implements the connection calculus [2,3], which is a calculus in the tableau family. Like other tableau methods the connection calculus is not limited to normal forms like clausal form, but unlike other tableau methods it is goal-directed. This means that inference steps are driven by complementary literals (in the sequent calculus these correspond to potential axioms), a feature which in general makes the connection calculus much more efficient than calculi that are driven by connectives, like analytic tableau calculi Goal-directed search is particularly powerful for problems that contain many axioms that are not required to prove the conjecture.

Although the connection method for FOL has been well documented, and implementations of it exist, specifying it in Maude is nevertheless non-trivial. Exploiting Maude's reflective properties we implement search strategies by explicitly operating on a stack of search states, abstracting from, e.g., details of the underlying data structures. While this level of abstraction is particularly useful for rapid prototyping, more optimized implementations should of course be sensitive to low-level details. Clearly, more low-level implementation details may also be exploited by search strategies.

Paper overview: Section 2 and 3 present the connection method, rewriting logic, and Maude. Section 4 and 5 consider the connection method and proof search in Maude. Section 6 discusses related work and Section 7 concludes.

2 The Connection Method: Paths through Matrices

This section briefly introduces the connection calculus. We assume a standard vocabulary; in particular, a *literal* is any atomic formula $P(t_1, \ldots, t_n)$ or its negation, and a *clause* is a conjunction of literals. A formula is in *prenex disjunctive normal form* (PDNF) if it is a closed formula of the form $\exists x_1 \ldots \exists x_n\, M$, where the *matrix*

M is a disjunction of clauses. It is well-known that any FOL formula A can be effectively transformed into a PDNF formula B, such that A is valid iff B is valid. Hereafter, we assume that all input formulas are on PDNF. This matrix representation exploits graphical metaphors. To illustrate the idea let $A_i,\ B_i, \ldots, R_i$ be literals. The PDNF formula

$$\exists x_1 \exists x_2 \ldots \exists x_n [(A_1 \wedge A_2 \ldots A_m) \vee (B_1 \wedge B_2 \ldots B_k) \vee \ldots \vee (R_1 \wedge R_2 \ldots R_j)]$$

is usually depicted as a matrix in which each clause has become a column:

$$\left[\begin{array}{ccccc} \left[\begin{array}{c} A_1 \\ A_2 \\ \vdots \\ A_m \end{array}\right] & \left[\begin{array}{c} B_1 \\ B_2 \\ \vdots \\ B_k \end{array}\right] & \ldots & \left[\begin{array}{c} R_1 \\ R_2 \\ \vdots \\ R_j \end{array}\right] \end{array}\right]$$

The existential quantifiers are implicitly present; all the variables inside literals can be seen as existentially quantified outside of the matrix. A *path* through the matrix is a set of literals, with one literal from each clause (or column).

A *connection* consists of two literals with the same predicate symbol and arity, but only one contains a negation sign. The connection is σ*-complementary* if the substitution σ unifies its two atomic formulas. A set of connections *spans* the matrix if all paths through the matrix contain a connection from the set.

Example 2.1 The formula $\forall x P(x) \rightarrow P(a) \wedge P(b)$ receives the matrix representation below to the left; the two paths through the matrix to the right.

$$\left[\begin{array}{cc} [\neg P(x)] & \left[\begin{array}{c} P(a) \\ P(b) \end{array}\right] \end{array}\right] \qquad \{\neg P(x), P(a)\} \qquad \{\neg P(x), P(b)\}$$

In this particular case the two paths comprise a spanning set of connections. However, no substitution can make both of the connections complementary.

Clauses in a matrix can be copied, in which case all free variables are replaced by fresh variables. The *multiplicity* μ for a matrix M is a function which assigns a positive integer to each clause in M; M^μ then results from M by, for each clause C, adding $\mu(C) - 1$ *free variable copies* of C to M. A PDNF formula with matrix M is *matrix provable* if there is a set of σ-complementary connections which spans M^μ, for a multiplicity μ and a substitution σ.

Matrix provability is a sound and complete characterization of validity in FOL [4]. For the matrix in Example 2.1 it is easy to see that one can increase the

multiplicity of the singleton clause to demonstrate matrix provability:

$$\left[\begin{array}{c} [\neg P(x)]\,[\neg P(y)] \end{array}\begin{bmatrix} P(a) \\ P(b) \end{bmatrix}\right]$$

Given the substitution $(x \leftarrow a, y \leftarrow b)$ both paths contain complementary connections. In this case the free variable copy adds little complexity as it only contains one element; this would not be the case if the clause were larger.

Proof search with the connection calculus is a connection-driven path-exploring process; the multiplicity is increased on demand and unification constrains the set of potentially closing substitutions. Complete strategies must fairly balance the incremental extension of partial paths (based on identification of new connections), the update of partial substitutions (with new unifiers), and the addition of copies of clauses. All implementations we know use iterative deepening, either on the number of inferences or on proof depth; i.e., the search space is explored up to a fixed multiplicity, which can gradually be increased. Our strategies will also use iterative deepening and backtracking.

3 Rewriting Logic, Reflection, and Maude

A rewrite theory is a 4-tuple (Σ, E, L, R) where the signature Σ defines the term language, E is a set of equations and membership sentences, L is a set of labels, and R is a set of labeled rewrite rules [12]. Rewrite rules apply to terms of given sorts (modulo equivalence), as specified in the membership equational logic theory (Σ, E). A rule $t \longrightarrow t'$ if cond allows a local instance of pattern t to become the corresponding instance of t', where cond consists of rewrites, equations, and memberships that must hold for the rule to apply.

Rewriting logic is *reflective* [6]; i.e., there is a finitely presented *universal* rewrite theory $\mathcal{U}$ in which any finitely presented rewrite theory $\mathcal{R}$ can be represented (including $\mathcal{U}$ itself). Let C and C' be configurations and $\mathcal{R}$ be a set of rewrite rules, represented in $\mathcal{U}$ as $\overline{C}$, $\overline{C}'$, and $\overline{\mathcal{R}}$, respectively. Denote by $\mathcal{R} \vdash C \rightarrow C'$ that C may be rewritten to C' in the rewrite theory $\mathcal{R}$. Using this notation, the equivalence $\mathcal{R} \vdash \mathcal{C} \rightarrow \mathcal{C}' \Leftrightarrow \mathcal{U} \vdash \langle \overline{\mathcal{R}}, \overline{\mathcal{C}} \rangle \rightarrow \langle \overline{\mathcal{R}}, \overline{\mathcal{C}'} \rangle$, states that if a term C in the rewrite theory $\mathcal{R}$ can be rewritten to a term C', then the meta-representation $\langle \overline{\mathcal{R}}, \overline{C} \rangle$ of C in $\mathcal{R}$ can be rewritten to the meta-representation $\langle \overline{\mathcal{R}}, \overline{C'} \rangle$ of C' in $\mathcal{R}$ in the universal rewrite theory $\mathcal{U}$.

Maude [5] is a tool for rewriting logic which includes facilities to meta-represent a theory $\mathcal{R}$ and to apply rules from $\mathcal{R}$ to the meta-representation of a term C using *descent functions*. Metalevel rewrite rules may be used to select which rule from $\mathcal{R}$ to apply to which subterm of C by defining a function which takes as arguments a finitely presented rewrite theory $\mathcal{R}$, a term C, and a deterministic strategy S. Further details on the theory and the use of reflection in rewriting logic and Maude may be found in [5,6].

```
sorts       FOLconstant FOLfunc FOLterm FOLtermlist Open Closed .
sorts       Lit LitSet Clause ClauseSet Matrix SearchState Success Failure .
subsorts    FOLconstant FOLfunc < FOLterm < FOLtermlist .
subsort     Nat < FOLconstant .
subsorts    FOLpredicate < Lit < LitSet .
subsort     Clause < ClauseSet .
subsorts    Open Closed < SearchState < SearchStateList .

op nil      : → FOLtermlist .
op _,_      : FOLtermlist FOLtermlist → FOLtermlist [ assoc id: nil prec 77 ] .
op _[_]     : Qid FOLtermlist → FOLfunc .
op _(_)     : Qid FOLtermlist → FOLpredicate .

op ¬_       : Lit → Lit .
op none     : → LitSet .
op _,_      : LitSet LitSet → LitSet [ assoc comm id: none ] .
op noClause : → Clause .
op [_]      : LitSet → Clause .
op none     : → ClauseSet .
op _,_      : ClauseSet ClauseSet → ClauseSet [ assoc comm id: none ] .
op [_]      : ClauseSet → Matrix .
```

Figure 1. The Maude specification of basic syntax.

4 The Connection Method in Maude: Basic Syntax

In the Maude implementation terms of FOL have sorts FOLconstant, FOLfunc, FOLterm, and FOLpredicate, cf. Fig. 1. Constants are represented by lower case letters, function and predicate symbols by quoted identifiers applied to lists of terms, and variables by standard Maude variables. We assume that formulas are in PDNF and define literals of sort Lit as (possibly negated) predicates. Literal disjunction and conjunction are implicitly given by the matrix representation. Clauses of sort Clause are sets of literals, a matrix of sort Matrix is a set of clauses. The matrix of Example 2.1 is represented by

$$\left[\left[\neg P(x)\right]\left[\begin{array}{c} P(a) \\ P(b) \end{array}\right]\right] \qquad \texttt{[[¬ 'P(X:FOLterm)], ['P(a), 'P(b)]]}$$

The implementation that we propose operates on terms of sort SearchStateList, cf. Fig. 2. A term of sort SearchState is of the form

⟨active path, active clause, remaining matrix, substitution, copy index⟩.

In the *initial state* ⟨none; noClause; M; none; 0⟩ of a search through a matrix M, the active path is empty, there is no active clause, the remaining matrix is M itself, the substitution is empty, and the copy index is zero. The ordering of literals in the active path is irrelevant, so the path is represented by a set of literals. Since connections depend on a variable substitution, the search state contains a

```
op ⟨_;_;_;_;_⟩ : LitSet Clause Matrix ObjSubstitution Int → SearchState .

mb open  stack: St:SearchStateList : Failure .
mb closed stack: nil               : Success .

op open      : → Open .
op closed    : → Closed .
op nil       : → SearchStateList .
op __        : SearchStateList SearchStateList → SearchStateList [ assoc id: nil ] .
```

Figure 2. The Maude specification of SearchState.

term of sort ObjSubstitution (with empty element none). This sort is an object level representation of Maude's sort Substitution; its terms associate terms of sort FOLterm to variables. The final argument of a search state is an index used to generate new variable names.

The connection method is formalized as a calculus by the rewrite rules of Fig. 3. Note incidentally that to avoid matching large lists of search states against the rewrite rules, we let the SearchState elements form a SearchStateList, which in turn is treated as a stack. This way the search procedure developed in the sequel needs only work with one SearchState element at a time. SearchState elements are pushed onto the stack as deductive rules generate new elements; if the current SearchState element is found to be connected, we pop a new element off the stack and proceed with that element. The proof search terminates when the stack is empty.

Rule *init* selects an active clause from the matrix of the initial state. Given an active clause, the search for connections can start. In rule *reductionRule*, there is a literal Lit1 in the active path and a complementary literal Lit2 in the active clause which are unifiable. In this case we remove the literal Lit2 known to contain a complementary connection from the active clause, eliminating further investigation of paths already known to be connected.

Rule *extensionRule* similarly compares a literal in the active clause to literals in the remaining matrix, which allows an eager pruning of the search space. This rule is a simplification rule; it can be simulated by *reductionRule* in combination with *extendPath*, so it is not necessary for completeness. However, *extensionRule* reduces the number of search states much more efficiently.

Rules *reductionRule* and *extensionRule* unify two literals (of opposite polarities). For this purpose we currently use Maude's built-in unification, by which a successful unification returns a substitution and a natural number. The latter is a variable index which can be passed between different unification problems in order to avoid name clashes in variable names generated by the substitutions. In the rules, unifyCompl(P,Q) is true if there is a unifier between the terms in literals P and Q, and false otherwise. In contrast, mgu constructs the most general unifier for the two literals, based on the substitution provided by Maude's unification. To distinguish free variables in a FOL term from variables constrained by unifiers, we conventionally denote by F(N):FOLterm and by B(N):FOLterm the free and bound variables indexed by N, respectively. The function newIndex provides a new vari-

able index, based on the two literals and the old index. The substitution is used to propagate variable substitutions in the calculus, based on a most general unifier for the free variables in literals. In the presentation, we omit the application of the substitution; the new unifier is applied to all literals of the initial matrix between rewrite steps.

Rule *extendPath* branches the search by adding a search state in which the active path is extended with a literal from the active clause. The other literals of the active clause remain in a separate search state. The accumulated unification is passed on to both search states. Two structural rules *closedPath* and *openPath* simplify the list of SearchState elements (they play the role of open and closed leaf nodes in the sequent calculus). Rule *closedPath* removes redundant search states from the SearchStateList, whereas rule *openPath* terminates the proof search in the case of a path with no connections. The latter rule should only be applied to the final and non-empty clause of a matrix. Provided that *extendPath* has only been applied when *extensionRule* is not applicable, no connection can be found for the active path at this stage (although the formula can still be valid.)

```
vars Lit1 Lit2 : Lit .                    var Cl : Clause .
vars Path LSet1 LSet2 : LitSet .          var ClSet : ClauseSet .
vars MGU Sub : ObjSubstitution .          vars N N2 : Int .

var St : SearchState .
var StL : SearchStateList .
op _ stack: _ : SearchState SearchStateList → Search .

rl [init]: ⟨none; noClause; [Cl, ClSet]; none; N⟩ stack: nil
⟹ ⟨none; Cl; [ClSet]; none; N⟩ stack: nil .

crl [reductionRule]: ⟨Path, Lit1; [Lit2, LSet1]; [ClSet]; Sub; N⟩ stack: StL
⟹ ⟨Path, Lit1; [LSet1]; [ClSet]; MGU; N2 ⟩ stack: StL
 if MGU := mgu(Lit1, Lit2, N) ∧ N2 := newIndex(Lit1, Lit2, N) ∧ unifyCompl(Lit1, Lit2) .

crl [extensionRule]: ⟨Path; [Lit1, LSet1];[[Lit2, LSet2],ClSet]; Sub; N⟩ stack: StL
⟹ ⟨Path, Lit1; [LSet2]; [ClSet]; MGU ; N2⟩
        stack: ⟨Path; [LSet1]; [[ Lit2 , LSet2], ClSet ]; Sub; N2⟩ StL
 if MGU := mgu(Lit1, Lit2, N) ∧ N2 := newIndex(Lit1, Lit2, N) ∧ unifyCompl(Lit1, Lit2) .

rl [extendPath]: ⟨Path; [Lit1, LSet1] ; [Cl, ClSet] ; Sub; N ⟩ stack: StL
⟹ ⟨Path, Lit1; Cl; [ClSet]; Sub; N⟩ stack: ⟨Path; [LSet1]; [Cl , ClSet ]; Sub; N⟩ StL .

rl [closedPath]: ⟨Path; [none]; [ClSet]; Sub; N⟩ stack: St StL ⟹ St stack: StL .

rl [closedPath]: ⟨Path; [none]; [ClSet]; Sub; N⟩ stack: nil ⟹ closed stack: nil .

rl [openPath]: ⟨Path; [Lit1, LSet1]; [none]; Sub; N⟩ stack: StL ⟹ open stack: nil .
```

Figure 3. The connection calculus in Maude with an explicit notion of stack.

Rule application must be strictly controlled to avoid unfair strategies. A case in point is *extendPath*, which should only be applied when the rules that prune the search space fail to apply. Such order constraint on the calculus will be handled by means of a rewrite *strategy*. In Maude, strategies can be implemented as meta-level rewrite theories applied to object-level theories, which is what we do in the next section.

5 Implementing the Proof Search

We present the main components of the implementation separately, leading to the formulation of a connection-based search procedure in Section 5.5. For simplicity, this procedure assumes a fixed multiplicity and does hence not implement iterative deepening. For a complete search procedure, all that is omitted in Section 5.5 is a function which gradually increases the multiplicity.

5.1 A Function for Case-based Rewriting

Let l be a Qid and applyRule$(\mathcal{M}, t, l)$ a function which applies a rule with label l to a term t in module M (simplifying Maude's metaApply by assuming that the rewrite may only occur at a unique position of t and ignoring the substitution). Following Maude conventions, this function returns a term of sort ResultTriple?; if the rule application succeeds it returns a term of the subsort ResultTriple. The standard function getTerm : ResultTriple → Term returns the term resulting from the rule application (the ResultTriple terms also includes a sort and a substitution). Let lab be a label, and L1 and L2 be lists of labels. A *case-based rewrite function* cases(M, T, L1) is defined as follows:

```
var M : Module . var T : Term .
vars L1 L2 : QidList . var lab : Qid .

op cases : Module Term QidList → Term .
op cases : Module Term QidList QidList → Term .

eq cases(M, T, L1) = cases(M, T, nil, L1) .
eq cases(M, T, L1, nil) = T .

ceq cases(M, T, L1, lab L2) =
  if (RESULT :: ResultTriple) then
        if (occurs(lab, 'reductionRule 'extensionRule 'extensionRule2))
        then cases(M, mguNewIndex(M, getTerm(RESULT)), nil, L1 lab L2)
        else cases(M, getTerm(RESULT), nil, L1 lab L2) fi
  else cases(M, T, L1 lab, L2) fi
if RESULT := applyRule(M, T, lab) .
```

The function cases(M, T, L1) repeatedly tries to apply the rules of module M to the term T in the order given by the labels in the list L1. When no rule is applicable, the

function terminates and the term is returned. The function mguNewIndex distributes information about variable bindings (mgu) and newly generated fresh variables, so other SearchState elements can update their newIndex and substitute variables. This is only relevant if the rewrite rule we are applying is a variable binding rule, hence the occurs check.

5.2 A Strategy for Basic Search

The basic search component defines a strategy where an active clause is selected by an *init* rule. Then, we recursively attempt to apply rules to the currently investigated SearchState element, respecting a given order and keeping track of substitutions and index values. Let gt and dt be wrapper functions for getTerm and downTerm. The strategy terminates once it locates a path without connections. In particular, it does not solve the matrix in Example 2.1.

```
var Mat : Matrix .   var Mo : Module .
var N : Nat .   var lab : Qid .
var L1 : QidList .

op init          : Matrix → Term .
op strategy      : Module Matrix QidList → Search .
op basicSearch : Matrix → Bool .

eq init (Mat)= upTerm(⟨none; noClause; Mat ; none; 0⟩ stack: nil) .

ceq strategy(Mo, Mat, lab L1) =
    if (RESULT :: ResultTriple) then dt(cases(Mo, gt(RESULT), L1))
    else open stack: nil fi      *** return member of Failure sort when init rule fails
 if RESULT := applyRule(Mo, init(Mat), lab) .

ceq basicSearch(Mat) = RESULT :: Success
 if RESULT := strategy(['FOL−CONNECTION], Mat,
    ' init 'reductionRule 'closedPath 'openPath 'extensionRule 'extendPath) .
```

5.3 Static Free Variable Copies

We now address clause copying. The function below implements *static copying*; i.e., copies are introduced before the proof search begins. The presented function makes a single copy of each clause with free variables, this can easily be modified by increasing the last parameter of the strategy function.

```
*** we extend init and strategy

op init      : Matrix Nat → Term .
op strategy : Module Matrix QidList Nat → Search .

eq init (Mat, N) =
```

```
upTerm(< none; noClause; staticCopy(Mat, N); none; sumFree(Mat) * N > stack: nil) .

ceq strategy(Mo, Mat, lab L1, N) =
    if (RESULT :: ResultTriple) then dt(cases(Mo, gt(RESULT), L1))
    else open stack: nil fi
  if RESULT := applyRule(M, init(Mat, N), lab) .

ceq staticCopySearch(Mat) = RESULT :: Success
 if RESULT := strategy(['FOL-CONNECTION], Mat,
    'init 'reductionRule 'closedPath 'openPath 'extensionRule 'extendPath, 1) .
```

Here, the auxiliary function sumFree : Matrix → Nat returns the number of free variables inside the Matrix, and staticCopy : Matrix Nat → Matrix returns the matrix with an additional N fresh copies of any clause that contains variables.

Example 5.1 These auxiliary functions work as follows:

```
red sumFree([['P(X:FOLterm), 'R(Z:FOLterm,X:FOLterm)], ['Q(a,b)]]) .
result NzNat: 2

red staticCopy ([[¬('P(X:FOLterm))], ['P(a),'P(b)]], 2) .
result Matrix: [[¬('P(X:FOLterm))], [¬('P(F0:FOLterm))],[¬('P(F1:FOLterm))],['P(a),'P(b)]]
```

5.4 Backtracking

A fair strategy requires that we keep track of the (number of) possible matches for a rule in a term and decide on an order for trying the different matches. For this purpose, we use the functionality provided by Maude's metaApply to apply a rewrite rule to the n'th matching position in a term. This way, we extend the case-based rewrite strategy above with backtracking support:

```
sorts BackTrack BackTrackList .
subsort BackTrack < BackTrackList .

op {_,_,_}      : Qid Term Nat → BackTrack .
op nil          : → BackTrackList .
op __           : BackTrackList BackTrackList → BackTrackList [assoc id: nil] .

op btrcases     : Module Term QidList QidList → Term .
op btrcases     : Module Term QidList QidList BackTrackList QidList → Term .

eq btrcases(M, T, L1, L2) = btrcases(M, T, nil, L1, nil, L2) .
eq btrcases(M, T, L1, nil, nil, L2) = T .

ceq btrcases(M, T, L1, lab L2, BTL, L3) =
 if (RESULT :: ResultTriple) then
   if (occurs(lab, L3))                    *** this is a rule we should backtrack over
   then btrcases(M, mguNewIndex(M, gt(RESULT)), nil, L1 lab L2, {lab, T, 1} BTL, L3)
```

```
   else btrcases(M, mguNewIndex(M, gt(RESULT)), nil, L1 lab L2, BTL, L3) fi
 else
      btrcases(M, T, L1 lab, L2, BTL, L3)
 fi
if RESULT := applyRule(M, T, lab) .

eq btrcases(M, T, L1, nil, { lab, T2, N } BTL, L3) =
 if(dt(T) :: Failure) then
  if(applyRule(M, T2, lab, N) :: ResultTriple)
  then btrcases(M, mguNewIndex(M, gt(applyRule(M, T2, lab, N))),
                   nil, L1, {lab, T2, N+1} BTL, L3)
  else
       btrcases(M, T, L1, nil, BTL, L3)
  fi
 else T fi .
```

If our current rule applies, the number of the next potential matching position is stored in the BackTrackList. Backtracking is then a matter of calling the search function with the elements of the backtrack-term, which are now stored in a list. Note that we can select the members of the label list; i.e., the rewrite rules to which backtracking should apply.

```
op strategy          : Module Matrix QidList QidList Nat → Search .
op backtrackSearch   : Matrix → Bool .

ceq strategy(Mo, Mat, lab L1, L2) =
   if(RESULT :: ResultTriple) then dt(btrcases(Mo, gt(RESULT), L1, L2))
   else open stack: nil fi
   if RESULT := applyRule(Mo, init(Mat), lab) .

ceq backtrackSearch(Mat) = RESULT :: Success
 if RESULT := strategy(['FOL−CONNECTION], Mat,
    'init 'reductionRule 'closedPath 'openPath 'extensionRule 'extendPath,
    'reductionRule 'extensionRule) .
```

5.5 Dynamic Free Variable Copies

In this section we consider a strategy for proof search in which the copies of clauses with free variables are added *dynamically* when deductive rules are applied. The main advantage of a dynamic scheme is that copies are created on demand. This limits the number of possible connections, in contrast to the a priori fixed number of copies provided by *staticCopySearch*. The idea is to make variables as *free* as possible, such that variable bindings occurring during unification affect as few variables as possible. For this purpose, clause copying will be associated with the rules which activate a new clause during the search.

```
rl [init2]: ⟨Path; noClause; [Cl, ClSet]; Sub; N⟩ stack: StL
⟹ ⟨Path; Cl; [copyClause(Cl, N), ClSet]; Sub; cntFree(Cl) + N⟩ stack: StL.

crl [extensionRule2]: ⟨Path; [Lit1, LSet1]; [[Lit2, LSet2], ClSet]; Sub; N⟩ stack: StL
⟹ ⟨Path,Lit1; [LSet2]; [ClSet]; MGU; N2⟩ stack: ⟨Path; [LSet1]; [Cl,ClSet]; Sub; N2⟩ StL
 if (unifyCompl(Lit1, Lit2)) ∧ MGU := mgu(Lit1, Lit2, N)
    ∧ N2 := newIndex(Lit1, Lit2, N) + cntFree([Lit2, LSet2])
    ∧ Cl := copyClause([Lit2, LSet2], newIndex(Lit1, Lit2, N)) .
```

First, the rule *init2* modifies the initialization rule (*init*) which selects the active clause from the initial matrix. The difference from the previous init rule is that a copy of the selected clause is placed in the remaining matrix. Recall that the function copyClause provides a fresh copy of the input clause, that newIndex provides a new variable index based on the two literals and the old index, and that the function cntFree counts the number of free variables in a clause. Next, we modify the rule extensionRule, which prunes the search by locating connections between elements in the active clause and the remaining matrix. Here, a fresh clause is generated and added to the second search state.

Example 5.2 The use of rule *extensionRule2* is illustrated as follows:

```
   ⟨Q(a); [P(X), U(c)]; [[¬P(b), R(Y)], [S(Z)]]; empty; 7⟩
⟹ ⟨Q(a), P(X); [R(Y)]; [[S(Z)]]; (X ← b); 8⟩
   ⟨Q(a); [U(c)]; [[¬P(b), R(F7)], S(Z)]; empty; 8⟩
```

The index of the SearchState elements is increased due to the free variable inside the clause that is copied. The main idea is that binding the variable Y should not affect the clauses left in the remaining matrix when the substitution is applied. Since a fresh copy of this clause is allowed, we replace the original clause with a copy to avoid name capture. Note that the original version of the clause could be left inside the remaining matrix as well. An iterative procedure where original versions of the clauses are left inside the remaining matrix is needed for completeness of the connection method.

By selecting the initial active SearchState element with rule *init2* and replacing *extensionRule* with *extensionRule2* in *basicSearch*, we get a search strategy *dynamicCopySearch* which incorporates a possible solution for dynamic fresh clause copies. (Note that there are several options for when to introduce copies; e.g., *extendPath* could also copy the activated clause.) In order to include backtracking over the *variable binding* rewrite rules, we define a strategy *dynamicCopyBacktrack* combining the backtracking strategy given in Section 5.4 with the new rule set. The strategy function is used once again, it should be noted that backtracking over the init-rules have not been presented, this is however only a minor detail to implement.

Search	SET044-5	SYN057-1	SYN005-1.010	SYN101-1.020.020	PUZ005-1
	4/8/6	5/13/4	11/10/20	17/37/24	51/112/56
basic	**S**, 8ms	**F**, 8ms	**S**, 24ms	**S**, 80ms	**F**, 6680ms
basic *	**S**, 20ms	**F**, 8ms	**S**, 24ms	**S**, 396ms	**F**, 61563ms
static copy	**S**, 28ms	**F**, 16ms	**S**, 36ms	**S**, 296ms	**F**, 11452ms
static copy *	**F**, 64ms	**F**, 16ms	**S**, 32ms	**S**, 21241ms	-
backtracking	**S**, 8ms	**F**, 1132ms	**S**, 24ms	**S**, 84ms	-
backtracking *	**S**, 24ms	**F**, 504ms	**S**, 20ms	**S**, 412ms	-
static copy, backtrack over init rule	**F**, 248ms	**S**, 40ms	**S**, 44ms	**S**, 312ms	-
dynamic copy	**F**, 8ms	**F**, 8ms	**S**, 24ms	**S**, 88ms	**S**, 224ms
dynamic copy, backtracking	**S**, 56ms	**S**, 80ms	**S**, 32ms	**S**, 88ms	**S**, 228ms

Figure 4. Sample results from strategy application. The *size* of each formula is suggested by its number of clauses/literals/variables (given in row 2 of the table).

```
ceq dynamicCopySearch(Mat) = RESULT :: Success
  if RESULT := strategy(['FOL−CONNECTION], Mat,
              ' init2 ' reductionRule 'closedPath 'openPath 'extensionRule2 'extendPath) .

ceq dynamicCopyBacktrack(M, ClSet) = RESULT :: Success
  if RESULT := strategy(['FOL−CONNECTION], Mat,
            ' init2 ' reductionRule 'closedPath 'openPath 'extensionRule2 'extendPath,
            ' init2 ' reductionRule 'extensionRule2) .
```

5.6 Comparison of Search Strategies

In order to compare search functions we apply the previously defined functions to a selection of formulas from the TPTP library [15]. The proof search was done on a laptop with a 1.7 GHz CPU and 1 Gb RAM running Linux. The results of the search strategy applications for some representative formulas are presented in Fig. 4. In the figure, **S** denotes that the proof search succeeded in proving the formula, and **F** that the search failed to prove the formula. As previously mentioned, extensionRule can be excluded. The three first proof searches have also been performed without this rule (marked with '∗'). We also considered a proof search where backtracking only applies to the init-rule and took one static copy of the free variable clauses prior to the proof search. In addition, the figure provides the time in milliseconds for each proof search to terminate. Open entries represent that the search did not produce any result within five minutes. Note that the formulas selected in Fig. 4 are all valid and easy to prove in state of the art systems. However, they also show that the behavior of the different strategies defined for the connection calculus are easy to compare using rewriting logic and Maude.

Most notable are perhaps the negative results produced by the different strategies, which allow the strengths of these simple strategies to be compared. The strategies are listed in the figure in increasing strength. Surprisingly, dynamicCopySearch has bad performance on **SET0044-5**; this is due to the choice of variable

binding and is solved by backtracking in dynamicCopyBacktrack. Although the basicSearch is obviously not complete it provides a useful insight into the time needed for efficient verification, due to its simplicity. Therefore, it may be interesting to compare strategies with more overhead to basicSearch. For the formulas in the figure, staticCopySearch performs surprisingly well. However, the significant overhead due to the initial extension of the matrix results in much slower results for large formulas such as PUZ005-1. The figure finally suggests that backtracking and the dynamic clause copies inserted during rule application adds little overhead. In particular, when backtracking or additional clause copies are not needed, the additional time needed with these strategies seems acceptable.

6 Related Work

A broad range of computational and deductive systems have been specified using rewriting logic and Maude; for examples, see [5]. In particular, the ITP tool is a theorem prover developed in Maude [7], exploiting reflection. In contrast to our automated proof search, ITP is an interactive prover developed for inductive reasoning about specifications in membership equational logic. A strategy language has been proposed for Maude [11,8] in which strategies may be composed using strategy combinators. The approach provides a nice separation of concerns between computation and control. Our strategies for *cases* and *btrcases* are examples of strategies which could potentially be expressed in this language. However our backtracking strategy allows the user to specify that backtracking applies to a specific subset of the rules of the rewrite theory, a feature which reduces the size of the search at runtime. Currently the strategy language is implemented at the meta-level and only supported by Full Maude. Thus, it does not fit directly with the connection calculus that we have presented here. However, a low-level integration of the strategy language with Maude will make it an interesting tool for further extensions of our work. Such an implementation is under development [8].

This work extends our previous work on the connection calculus for propositional logic [10]. We are not aware of any attempts to design a theorem prover for FOL in Maude using a system with a level of sophistication that is comparable to the connection calculus. The system and strategy we implement in this paper is closely related to the Prolog theorem prover leanCoP [14]. This is an implementation which takes full advantage of Prolog's backtracking and unification scheme, a feature which allows extremely compact code, but also makes it more difficult to control backtracking.

The tableaux workbench [1] and LoTREC [9] are initiatives that are similar to ours, in the sense that they support high-level specification of proof systems and strategies. They are, however, limited to tableau methods, which are not goal-directed, and their strategy languages are much more restricted than what Maude's meta-level provides.

7 Conclusion and Future Work

The paper presents a rewriting logic approach to the implementation of a connection-driven search engine for FOL. A rewrite theory is defined for the connection method, and variations of search strategies are explored at the Maude meta-level. This facilitates comparison and experimentation with strategies for proof search, as these control the same underlying rewrite theory.

The motivation behind the current work is to develop a deduction platform in Maude that supports flexible strategies. In future work we enable incorporation of contextual knowledge about assumption sets into search procedures, a feature which supports the design of *special-purpose* theorem provers. We also intend to operate both on a meta-level (the usual level of strategies) and at a meta-meta-level, the latter in order to select, refine and compose new strategies at run-time. This *adaptive* behaviour can be guided by information about the search history in addition to the present configuration of the search.

Our long-term perspective is to contribute to the design of efficient domain-specific reasoning algorithms for expressive non-classical logics. In contrast to, say, the formalization of mathematical reasoning, many applications are characterized by a large set of premises with fairly shallow logical structure. For example, ontologies with more than 100 000 concepts are not unusual, often with simple concept definitions. We believe that a goal-directed method like the connection method has potential in such contexts, along with domain-specific search strategies. The fact that the connection method does not require any normal form makes it attractive also for non-classical logics. For intuitionistic logic, for instance, the connection-based theorem prover ileanCoP by far outperforms any other implementation [13]. Matrix characterisations already exist for a number of non-classical logics [16], and a future ambition is to gradually extend the current work to more sophisticated logics and more complex strategies.

Acknowledgement

We are grateful to Jens Otten for interesting discussions on proof search for the connection method and to Steven Eker for giving us access to Maude alpha-versions with unification support. This paper was written while Einar Broch Johnsen was enjoying the hospitality of the United Nations University - International Institute for Software Technology in Macau.

References

[1] P. Abate and R. Goré. The tableaux work bench. In *Proc. TABLEAUX 2003*, LNCS 2796, pages 230–236. Springer, 2003.

[2] P. B. Andrews. Refutations by matings. *IEEE Trans. Computers*, 25(8):801–807, 1976.

[3] W. Bibel. An approach to a systematic theorem proving procedure in first order logic. *Computing*, 12:43–55, 1974.

[4] W. Bibel. *Automated Theorem Proving*. Vieweg, Wiesbaden, 2nd edition, 1987.

[5] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada. Maude: Specification and programming in rewriting logic. *Theoretical Computer Science*, 285:187–243, Aug. 2002.

[6] M. Clavel and J. Meseguer. Reflection in conditional rewriting logic. *Theoretical Computer Science*, 285:245–288, Aug. 2002.

[7] M. Clavel, M. Palomino, and A. Riesco. Introducing the ITP tool: a tutorial. *Journal of Universal Computer Science*, 12(11):1618–1650, 2006.

[8] S. Eker, N. Martí-Oliet, J. Meseguer, and A. Verdejo. Deduction, strategies, and rewriting. In *Proc. STRATEGIES 2006*, ENTCS 174: 3–25. Elsevier, July 2007.

[9] O. Gasquet, A. Herzig, D. Longin, and M. Sahade. LoTREC: Logical tableaux research engineering companion. In *Proc. TABLEAUX 2005*, LNCS 3702, pages 318–322. Springer, 2005.

[10] B. Holen, E. B. Johnsen, and A. Waaler. Representing strategies for the connection calculus in rewriting logic. In *Proc. FTP 2005*, pages 130–141, Aug. 2005. Tech. rep. 13/2005, Inst. für Informatik, Universität Koblenz-Landau.

[11] N. Martí-Oliet, J. Meseguer, and A. Verdejo. Towards a strategy language for Maude. In *Proc. WRLA 2004*, ENTCS 117: 417–441. Elsevier, Jan. 2005.

[12] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96:73–155, 1992.

[13] J. Otten. Clausal connection-based theorem proving in intuitionistic first-order logic. In *Proc. TABLEAUX 2005*, LNCS 3702, pages 245–261. Springer, 2005.

[14] J. Otten and W. Bibel. leanCoP: lean connection-based theorem proving. *Journal of Symbolic Computation*, 36(1–2):139–161, 2003.

[15] G. Sutcliffe and C. Suttner. The TPTP Problem Library: CNF Release v1.2.1. *Journal of Automated Reasoning*, 21(2):177–203, 1998.

[16] A. Waaler. Connections in nonclassical logics. In J. A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, pages 1487–1578. Elsevier and MIT Press, 2001.