Automatic Proof Checking and Proof Construction by Tactics

Gilles Dowek

Automatic Proof Checking and Proof Construction by Tactics

Notes for the Workshop on Meta-Variables, September 1991.

Gilles Dowek

INRIA[*†]

In this note we compare two kinds of systems that verify the correctness of mathematical developments: *proof checking* and *proof construction by tactics* and we propose to merge them in a single system. We consider mathematics formalized in the Calculus of Constructions [3] [4] but the ideas given here are not specific to this formalism. As an example of a proof checking system we consider the *constructive engine* of the system Coq [11] and as an example of a proof construction by tactics system we consider the *proof assistant* of the system Coq [7] designed in the spirit of the system LCF [8].

1 Motivations

1.1 The Constructive Engine and the Proof Assistant

The Constructive Engine and the Proof Assistant are both systems that aim to give a tool to a mathematician to write a theorem and its proof and check the correctness of the proof.

With the *constructive engine* the user gives the proof of the theorem as a λ-term. The answer of the machine is a binary information: *right* or *wrong* according to the correctness of the proof. For instance, the following developments are correct.

Example 1:

```
Theorem I.
Statement (P:Prop)(P -> P).
Proof [P:Prop][x:P]x.
```

Example 2:

```
Parameter T:Prop.

Parameter R:T -> T -> Prop.

Parameter Eq:T -> T -> Prop.

Axiom Antisym:(x:T)(y:T)(R x y) -> (R y x) -> (Eq x y).

Parameter a:T.

Parameter b:T.
```

[*]B.P. 105, 78153 Le Chesnay CEDEX, France. dowek@margaux.inria.fr

[†]This research was partly supported by ESPRIT Basic Research Action "Logical Frameworks".

```
Axiom ax1:(R a b).

Axiom ax2:(R b a).

Theorem th.
Statement (Eq a b).
Proof (Antisym a b ax1 ax2).
```

With the *proof assistant* the machine is more active, the theorem to be proven is taken as a *goal*. The set of goals is transformed by *tactics*, a tactic is a function that maps a set G of goal to a set G' of goals such that proof of the goals of G can be constructed from proofs of the goals of G'. Proving a proposition consists in applying tactics until we get an empty set of goals. The same examples are written as follows.

Example 1:

```
Goal (P:Prop)(P -> P).

Intro.
                    1 subgoal
                      P->P
                      ============================
                        P : Prop

Intro.
                    1 subgoal
                      P
                      ============================
                        H : P
                        P : Prop

Apply H.
                    Goal proved!
```

Example 2 (we keep the same declarations of parameters and axioms):

```
Goal (Eq a b).

Apply Antisym.
                      (R a b)
                    subgoal 2 is:
                      (R b a)

Apply ax1.
                      (R b a)
```

```
Apply ax2.
                    Goal proved!
```

Let us consider the first example. We first have to prove the proposition $(P : Prop)(P \to P)$, the tactic *Intro* transforms this goal into the goal $P \to P$ in a context where P is a proposition then the same tactics transforms the goal in P in a context where P is a proposition and H a proof of P. Then the tactic *Apply* H finds the proof H for the goal P and the set of remaining subgoal is empty, so the proof is over.

Then from this sequence of tactics we can build the proof $[P : Prop][H : P]H$. This proof is built in three steps: when we first apply the tactic *Intro* we build the proof $[P : Prop]?y$ where $?y$ is a meta-variable denoting the forthcoming proof of the current goal $P \to P$. When we apply another time the tactic intro, we build the proof $[P : Prop][H : P]?z$ where $?z$ is a meta-variable denoting the forthcoming proof of the current goal P. When we apply the tactic *Apply* H we get the proof $[P : Prop][H : P]H$.

When the goal is proved, the theorem can be added to the context of the engine with the command:

```
Save I.
```

The symbols $?y$, $?z$ are called meta-variables, they denote proof-terms to be constructed. When such a proof term is constructed it is substituted to the meta-variable, remark that this substitution may capture variables for instance when we substitute H to $?z$ in the term $[P : Prop][H : P]?z$ we get the term $[P : Prop][H : P]H$.

In a proof assistant, the tactic *Intro* and the tactic *Apply* can be distinguished as basic tactics. The tactic *Intro* transforms the goal $(x : A)B$ in the goal B adding $x : A$ to the context. The tactic *Apply* uses a known proposition $(x_1 : P_1)...(x_n : P_n)P$ and a goal Q, unifies P and Q (this unification binds some of the x_i's), and gives back as subgoals the P_i's such that x_i is not bound by unification. For instance in the second example we want to prove $(Eq\ a\ b)$ we unify $(Eq\ x\ y)$ and $(Eq\ a\ b)$ and we generate the subgoals $(R\ a\ b)$ and $(R\ b\ a)$ the proof associated to this step is $(Antisym\ a\ b\ ?v\ ?w)$ where $?v$ and $?w$ are proofs of $(R\ a\ b)$ and $(R\ b\ a)$ to be found.

1.2 Merging Proof Checking and Proof Construction by Tactics

Most of mathematics verifying systems as for instance the system Coq [7] can be used in two modes: proof checking and proof construction by tactics. We design a system with a single mode and such that:

- The declaration of the statement of a theorem:

```
Theorem I.
Statement (P:Prop)(P -> P).
```

and of a goal:

```
Goal (P:Prop)(P -> P).
```

are done in the same way.

- The user can work on a proof, go to another one come back to the first, etc.

• The user can work by tactics (top-down) on a goal, decide to stop to prove a lemma (bottom-up) and go back to its proof.

• Since unification is undecidable we do not want the tactic *Apply* to be restricted by a subcase of unification (as first order unification) but we want to let the user guide the machine in the unification tree as he does in the proof tree.

2 Meta-variables are Variables: An Engine

2.1 Meta-variables are Variables

When we have considered incomplete proofs as $(Antisym\ a\ b\ ?v\ ?w)$ we have added to the formalism of terms symbols $?v$, $?w$ that denotes some terms. In the same way, in elementary algebra when we can write an equation as $x^2 + 6 = 5x$ we add to the formalism of integers a symbol x that denotes an integer. The expression $x^2 + 6$ is an incomplete integer. The symbol x is called a variable.

In contrast with the formalism of numbers, the λ-calculus already has a notion of variable. So adding a new notion of meta-variables is not useful and we can write this incomplete proof $(Antisym\ a\ b\ v\ w)$. As far as term formation and type checking are concerned the variables v and w are not different from a and b, in particular v and w *have to be declared* in a context Γ where $(Antisym\ a\ b\ v\ w)$ is well-formed.

The variables v and w are distinguished from a and b only for substitution since v and w can be instanciated and a and b cannot. This information is expressed in adding a quantifier to each variable in the context, the universal quantifier $\forall$ is added to the declaration of variables that cannot be instanciated (as a and b) and the existential quantifier $\exists$ is added to the declaration of the variables that can be instanciated (as v and w).

2.2 Constraints

When we apply a substitution to a variable x the term substituted to x must have the same type as x. Let us consider the context:

$$\Gamma = [\forall A : Prop; \exists X : Prop; \exists y : X \to A]$$

a priori we cannot substitute the term $[x : X]x$ to the variable y since the term $[x : X]x$ has type $X \to X$ and y has type $X \to A$. But if we apply first the substitution $X \leftarrow A$, the substitution $y \leftarrow [x : A]x$ is allowed.

So before applying the substitution $y \leftarrow [x : X]x$ we should unify the type of y and the type of $[x : X]x$. Since unification is in general a difficult problem, we do not want to have to perform a unification step before each substitution, so we allow the substitution $y \leftarrow [x : X]x$ but we will keep a constraint $X \to X = X \to A$ to remind that this equation has to be solved even if we not want to solve it now.

In the same way, when we have an existential variable (a goal) $g : (Eq\ a\ b)$ we may introduce new existential variables $h_1 : T$, $h_2 : T$, $h_3 : (R\ h_1\ h_2)$, $h_4 : (R\ h_2\ h_1)$ and instanciate the variable g the term $(Antisym\ h_1\ h_2\ h_3\ h_4)$. The generated constraint is $(Eq\ a\ b) = (Eq\ h_1\ h_2)$. We do not have to solve this equation to perform the substitution, but this equation will restrict the substitutions to be performed for h_1 and h_2 in the future.

If, as in this case, the constraint is a first order unification problem it can be solved and the substitution $h_1 \leftarrow a, h_2 \leftarrow b$ can be performed. But if the constraint is a higher order unification problem then the user can propose substitutions for the variables occurring in the equation and guide the machine in the unification tree as he does in the proof tree[1].

2.3 Constrained Quantified Contexts

Definition: Constrained Quantified Contexts

A *quantified declaration* is a triple $< Q, x, T >$ (written $Qx : T$) where Q is a quantifier ($\forall$ or $\exists$), x a symbol and T is a term. A *constant definition* is a triple $< x, t, T >$ (written $x := t : T$) where x is a symbol and t and T terms. A *constraint* is a pair of terms $< a, b >$ (written $a = b$). A *constrained quantified context* is a list $\Gamma = [e_1; ...; e_n]$ such that e_i is either a quantified declaration a constant definition or a constraint. These constrained quantified contexts are generalization of Miller's *mixed prefixes* [12] which are lists of quantified declarations.

Non constrained, non quantified contexts are identified with constrained quantified contexts with only universal variables and constants.

Definition: Equivalence Modulo Constraints

Let Γ be a constrained quantified context, we define the relation between terms $\equiv_\Gamma$ as the smallest equivalence relation compatible with terms structure such that:

- if $t \equiv t'$ then $t \equiv_\Gamma t'$,
- if $(a = b) \in \Gamma$ then $a \equiv_\Gamma b$.

Definition: Typing Rules

First we modify the rules of the Calculus of Constructions to deal with the new syntax:

The rule:

$$\frac{\Gamma \vdash T : s}{\Gamma[x : T] \text{ well-formed}}\, s \in \{Prop, Type\}$$

is replaced by:

$$\frac{\Gamma \vdash T : s}{\Gamma[Qx : T] \text{ well-formed}}\, s \in \{Prop, Type\}$$

and we add the rule:

$$\frac{\Gamma \vdash a : T \quad \Gamma \vdash b : T}{\Gamma[a = b] \text{ well-formed}}$$

Then we extend the system by replacing the rule:

$$\frac{\Gamma \vdash T : s \quad \Gamma \vdash T' : s \quad \Gamma \vdash t : T \quad T \equiv T'}{\Gamma \vdash t : T'}\, s \in \{Prop, Type\}$$

by:

$$\frac{\Gamma \vdash T : s \quad \Gamma \vdash T' : s \quad \Gamma \vdash t : T \quad T \equiv_\Gamma T'}{\Gamma \vdash t : T'}\, s \in \{Prop, Type\}$$

[1] This view of unification as constrained resolution leads in [6] to a complete proof synthesis method where unification and resolution are merged in a single algorithm.

This defines two new judgements: Γ is *well-formed using the constraints* and t *has type* T *in* Γ *using the constraints*.

Remark: A term may be well-typed in Γ using the constraints and still be not normalizable.

Definition: Well-typed Without Using the Constraints

Let Γ be a context and t and T be two terms. The term t is said to be *of type* T *in* Γ *without using the constraints* if there exists Δ subcontext of Γ (i.e. obtained by removing some items of Γ) such that Δ has no constraints, is a well-formed context and $\Delta \vdash t : T$.

Proposition: If a term is well-typed in a context without using the constraints then it is strongly normalizable.

Definition: Normal Form of a Context

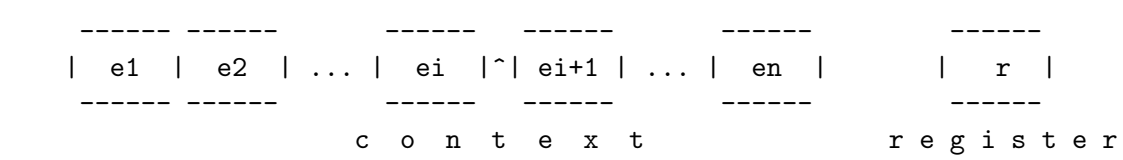
Let Γ be a context, the *normal form* of Γ is obtained by putting all the types of variables that are well typed without the constraints and all the constraints which terms are well-typed without the constraints in β-normal η-long form.

Definition: Success and Failure Contexts

A normal context Γ is said to be *a success context* if it has only universal variables, and constraints relating identical terms. It is said to be *a failure context* if there is a constraint relating two normal well-typed ground terms which are not identical.

2.4 An Extended Constructive Engine

We consider an extension of the constructive engine [11]. This machine has a register that contains a term and a context which is a quantified constrained context. The context is separated in two parts by an index:



Two invariants are maintained: the context $[e_1; ...; e_i; e_{i+1}; ...; e_n]$ is a well-formed context and the term r is well-typed in the context $[e_1; ...; e_i]$.

The basic instructions of the machine are:

• move the index on the left or to the right and erase the register,

• check that a term t is well typed in the context $[e_1; ...; e_i]$ and put it in the register.

• check that the type of the term in the register is $Prop$ or $Type$ and insert a declaration of a new variable of this type between e_i and e_{i+1}.

• check that e_i is the declaration of an existential variable y, erase it, insert a constraint $T = U$ where T is the type of this existential variable and U the type of the term r, insert a constant declaration $y := r$, put the context in normal form and remove the constraints relating equal terms and fail if the context is a failure context.

In a more sophisticated engine the constraints that are first order unification problems are automatically solved. This can be done in simplifying the constraints [9] [10] [6] and solving the

trivial equations $x = t$ where x is an existential variable[2].

2.5 Proof Checking with this Engine

When we verify a development as:

```
Theorem I.
Statement (P:Prop)(P -> P).
Proof [P:Prop][x:P]x.
```

we construct the term $(P : Prop)(P \to P)$ we declare an existential variable I of this type, we construct the term $[P : Prop][x : P]x$ and we instanciate the variable y with this term, the constraint $(P : Prop)(P \to P) = (P : Prop)(P \to P)$ is generated and removed since it relates equal terms (if the term had not the good type, we would get a failure context) and a constant $I := [P : Prop][x : P]x : (P : Prop)(P \to P)$ is added to the context.

2.6 Proof Construction by Tactics with this Engine

2.6.1 Application

The main idea in this section is that applying a tactic is applying a substitution to the current context. For instance let us consider the context:
$\Delta = [\forall T : Prop; \forall R : T \to T \to Prop; \forall Eq : T \to T \to Prop;$
$\forall Antisym : (x : T)(y : T)(R\ x\ y) \to (R\ y\ x) \to (Eq\ x\ y); \forall a : T; \forall b : T; \forall ax1 : (R\ a\ b); \forall ax2 : (R\ b\ a)]$ and $\Gamma = \Delta[\exists x : (Eq\ a\ b)]$. We declare new existential variables $h_1 : T$, $h_2 : T$, $h_3 : (R\ h_1\ h_2)$, $h_4 : (R\ h_2\ h_1)$.

We perform the substitution:

$$x \leftarrow (Antisym\ h_1\ h_2\ h_3\ h_4)$$

We get the context:
$\Delta[\exists h_1 : T; \exists h_2 : T; \exists h_3 : (R\ h_1\ h_2); \exists h_4 : (R\ h_2\ h_1); (Eq\ h_1\ h_2) = (Eq\ a\ b)]$
The constraint is a first order unification problem, it can be solved automatically, i.e. the following substitutions can be performed:

$$h_1 \leftarrow a$$

$$h_2 \leftarrow b$$

We get the context $\Delta[\exists h_3 : (R\ a\ b); \exists h_4 : (R\ b\ a)]$. We have this way performed a *Apply* step.

If the constraint on h_1 and h_2 were more complicated we would have proposed substitutions for these variables, so the tactic *Apply* is not limited by a unification algorithm.

2.6.2 Introduction

A problem with this engine is that the introduction tactic is not the mere application of a substitution to a context, because we have to take into account that when we instanciate h by $[x : T]k$ the variable k may be substituted by a term where x occurs. In the second part of this note we generalize the engine in order to be able to use this tactic.

[2]Also the argument-restricted unification problems [12] [13] [14] can be solved, if we simplify equations and solve the trivial equations $(x\ c_1\ ...\ c_n) = t$ where x is an existential variable and $c_1, ..., c_n$ are atomic terms which heads are distinct universal variables declared on the right of x.

2.6.3 Explicit Dependencies

Let us consider a goal $(x_1 : P_1)...(x_n : P_n)P$

When we apply n times the tactic $Intro$ and then once the tactic *Apply* with a head u we get the incomplete proof $[x_1 : P_1]...[x_n : P_n](u\ h_1\ ...\ h_p)$ where the variables $h_1, ..., h_p$ must be instanciated by terms where the variables $x_1, ..., x_n$ may occur. We may mark explicitly this dependence in anti-skolemizing these variables and considering the term:

$$[x_1 : P_1]...[x_n : P_n](u\ (h_1\ x_1...x_n)\ ...\ (h_p\ x_1\ ...\ x_n))$$

now the variables $h_1, ..., h_p$ must be instanciated by terms well formed in the same context as the initial goal.

This anti-skolemization has to be done after n introductions and one application. It cannot be done after one introduction because it would give the substitution $h \leftarrow [x : T](k\ x)$ i.e. modulo η-conversion to the substitution $h \leftarrow k$.

Proof synthesis methods using this elementary substitution (and also more general substitutions) are studied in [6].

This elementary substitution is not very usable for interactive proof construction since subgoals appear with unuseful and confusing dependencies. We are going now to generalize the engine to permit an introduction tactic.

3 Introduction: An Engine with Sections

3.1 Sections and Discharge Operation

Sections have been introduced in the framework of proof verification [1] [2] (see also [5]) to simplify the way proofs are written. Instead of writing:

```
Theorem I.
Statement (P:Prop)(P -> P).
Proof [P:Prop][x:P]x.
```

We write:

```
Theorem I.
   Variable P:Prop.
   Hypothesis x:P.
Statement P.
Proof x.
```

or:

```
Section I.
   Variable P:Prop.
   Hypothesis x:P.

   Theorem I.
   Statement P.
   Proof x.
End I.
```

When such a development is checked, the variable P and x are first declared. Then the theorem $I := x : P$ defined and *when the section is closed* the variables P and x are erased from the context and discharged in the constant I to make $I := [P : Prop][x : P]x : (P : Prop)(P \to P)$. So the successive values of the context are:
$[\,]$
$[\forall P : Prop]$
$[\forall P : Prop; \forall x : P]$
$[\forall P : Prop; \forall x : P; I := x : P]$
$[I := [P : Prop][x : P]x : (P : Prop)(P \to P)]$

Extra information must be added in the context to remind the limits of sections and the locality or globality of objects.

3.2 Explicit Sections

Another way to implement sections is to add two new kind of items in the context: beginnings and ends of sections. Checking the same development, the context would take the values:
$[\,]$
$[Begin]$
$[Begin; \forall P : Prop]$
$[Begin; \forall P : Prop; \forall x : P]$
$[Begin; \forall P : Prop; \forall x : P; I := x : P]$
$[Begin; \forall P : Prop; \forall x : P; I := x : P; End]$

Now when we want to access to the constant I we remark that it is inside a closed section where there are two local variables so we discharge these variables in the constant and we get $I := [P : Prop][x : P]x : (P : Prop)(P \to P)$. We cannot access to the variables P and x since they are local to a closed section and so they are out of scope.

In the first method the discharge operation is made once when the section is closed and access to a variable is simple. In the second method closing a section is very simple since we just have to add an item *End* to the context, but the access to a variable or a constant is complicated since we have to discharge local variable in it at each access. This can be compared with the call by value or call by name evaluation strategy.

3.3 Introduction Tactic

In an engine with explicit sections it is possible to have an introduction tactic. Each existential variable is a little section. For instance, let us consider the context:

$$[Begin; \exists h : (P : Prop)(P \to P); End]$$

We perform an introduction and get:

$$[Begin; \forall P : Prop; \exists h : P \to P; End]$$

From the outside of the section, the variable P cannot be seen and h has type $(P : Prop)(P \to P)$ but from the inside of the section P can be seen and h has type $P \to P$. If we perform another introduction we get:

$$[Begin; \forall P : Prop; \forall x : P; \exists h : P; End]$$

Then an application:

$$[Begin; \forall P : Prop; \forall x : P; h := x : P; End]$$

Seen from the outside of the section the value of h is $h := [P : Prop][x : P]x : (P : Prop)(P \to P)$

3.4 Physical Closing of an Explicit Section

When a section does not contain any existential variable it can be physically closed i.e. the context can be replaced by the equivalent context:

$$[h := [P : Prop][x : P]x : (P : Prop)(P \to P)]$$

4 Examples

A prototype of this system is implemented in an experimental version of the system Coq. We give here two proofs of a lemma of Tarski's theorem. The first is written completely top-down. In the second many lemmas are given such that the proof of each of them is very short.

```
Parameter T:Prop.

Parameter Eq:T->T->Prop.

Parameter R:T->T->Prop.

Axiom Antisym.
Assumes (x:T)(y:T)(R x y) -> (R y x) -> (Eq x y).

Axiom Trans.
Assumes (x:T)(y:T)(z:T)(R x y) -> (R y z) -> (R x z).

Parameter f:T->T.

Axiom Incr.
Assumes (x:T)(y:T)(R x y) -> (R (f x) (f y)).

Definition Pre = [x:T](R x (f x)).

Parameter M:T.

Axiom Up.
Assumes (x:T) (Pre x) -> (R x M).

Axiom Least.
Assumes (y:T)((x:T)(Pre x)->(R x y))->(R M y).
```

Example 1:

```
Theorem Tarski.

Statement (Eq M (f M)).

Apply Antisym.
Apply Up.
Apply Incr.
Apply Least.
Intro.
Intro. (* x13 *)
Apply Trans.
Apply Incr.
Apply Up.
Apply x13.
Apply x13.
Apply Least.
Intro.
Intro. (* x27 *)
Apply Trans.
Apply Incr.
Apply Up.
Apply x27.
Apply x27.
```

Example 2:

```
Theorem Tarski.

Statement (Eq M (f M)).

      Remark One.

         Variable y:T.

         Hypothesis v.
         Assumes (Pre y).

      Statement (R y (f M)).

         Remark Rem.
         Statement (R y M).
         Apply Up. Apply v.

         Remark Rem'.
         Statement (R (f y) (f M)).
         Apply Incr. Apply Rem.
```

```
      Apply Trans. Apply Rem'. Apply v.

      Remark Two.
      Statement (R M (f M)).
      Apply Least. Assumption One.

      Remark Three.
      Statement (R (f M) (f (f M))).
      Apply Incr. Apply Two.

      Remark Four.
      Statement (R (f M) M).
      Apply Up. Apply Three.

Apply Antisym. Apply Four. Apply Two.
```

References

[1] N.G. de Bruijn, The Mathematical Vernacular, A Language For Mathematics With Typed Sets, *Proceedings of the Workshop on Programming Logic*, Marstrand, Sweden, 1987.

[2] N.G. de Bruijn, The Mathematical Vernacular: Examples, Unpublished manuscript.

[3] Th. Coquand, Une Théorie des Constructions, *Thèse de troisième cycle*, Université Paris VII, 1985.

[4] T. Coquand, G. Huet, The Calculus of Constructions, *Information and Computation*, 76, 1988, pp. 95-120.

[5] G. Dowek, Naming and Scoping in a Mathematical Vernacular, *Rapport de Recherche 1283*, INRIA, 1990.

[6] G. Dowek, A Complete Proof Synthesis Method for Type Systems of the Cube.

[7] G. Dowek, A. Felty, G. Huet, H. Herbelin, Ch. Paulin-Mohring, B. Werner, The System Coq User's Guide, INRIA 1991.

[8] M.J. Gordon, A.J. Milner, C.P. Wadsworth, *Edinburgh LCF*, Lecture Notes in Computer Science 78, Springer-Verlag, 1979.

[9] G. Huet, A Unification Algorithm for Typed λ-calculus, *Theoretical Computer Science*, 1, 1975, pp. 27-57.

[10] G. Huet, Résolution d'Équations dans les Langages d'Ordre 1,2, ..., ω, *Thèse de Doctorat d'État*, Université de Paris VII, 1976.

[11] G. Huet, The Constructive Engine, *A Perspective in Theoretical Computer Science*, Commemorative Volume for Gift Siromoney, R. Narasimhan (Ed.), World Scientific Publishing, 1989.

[12] D. A. Miller, Unification Under a Mixed Prefix, To appear in *Journal of Symbolic Computation*.

[13] D. A. Miller, A Logic Programming Language with Lambda-Abstraction, Function Variables, and Simple Unification *Extension of Logic Programming*, P. Schroeder-Heister (Ed.), Lecture Notes in Computer Science 475, Springer-Verlag, 1991, pp. 253-281. Also Report ECS-LFCS-91-159, University of Edinburgh, 1991. Also to appear in Journal of Logic and Computation.

[14] F. Pfenning, Unification and anti-Unification in the Calculus of Constructions, To appear in *Proceedings of Logic in Computer Science*, 1991.