Automated Deduction and Algebra
Lecture I

Michael K. Kinyon

Department of Mathematics

University of Lisbon, 8 June 2009

Automated Theorem Provers

In these lectures, we will use Prover9, developed by William McCune. There are other good provers available. They all have advantages and disadvantages.

- Waldmeister
  - Equational reasoning only
  - Can solve problems “out of the box” that Prover9 cannot (and vice versa)
  - Not hard to use (simple syntax), but not flexible in its settings
- Vampire, E, Equinox, Others
  - Multipurpose, strong in many areas
  - Can solve problems Prover9 cannot (and vice versa)
  - Sometimes obscure syntax, tricky to use

Finite Model Builders

In these lectures, we will use Mace4, also developed by McCune. There are not as many finite model builders as there are provers, but here are a couple of them.

- SEM
  - Still used, but mostly obsolete
- Paradox
  - Powerful for logic problems, can find large models in that setting
  - For equational problems, no better than others

Where to get it

Prover9, Mace4 and supporting software can all be downloaded from:

`http://www.cs.unm.edu/~mccune/prover9/`

There is a GUI (graphical user interface) version and a command line version. We will mostly use the GUI here, but in some ways, the command line version is more flexible.

Assumptions and goals

The two main lists in a Prover9 (or Mace4) input file are the *assumptions* list and the *goals* list. These are delimited by

```
formulas(assumptions).
...
end_of_list.
```

and

```
formulas(goals).
...
end_of_list.
```

(All lists end with `end_of_list.`)

Connectives

The default basic logical connectives are:

–	not
|	or
&	and
–>	implies
<–>	if and only if

In equality problems, one can use != for “not equal”.
Thus “a != b” is the same as “-(a = b)”.

Formulas

An *atomic formula* is an *n*-ary predicate symbol applied to *n* terms is an atomic formula.
For example, an equation is an atomic formula; equality is a 2-term predicate.

Formulas are defined recursively:

- An atomic formula is a formula,
- if *F* and *G* are formulas, then so are:
  –*F*, *F*|*G*, *F*&*G*, *F* –> *G*, *F* <–> *G*.
- if F is a formula and x is a variable, then these are formulas:
  all *x F* exists *x F*

Literals and Clauses

A *literal* is either an atomic formula or the negation of an atomic formula.

A *clause* is a formula consisting of a disjunction of literals. All variables in a clause are assumed to be universally quantified.

For instance, an equation, which is just an atomic formula, is a clause:

```
x * (y * (x * (y * z))) = x * (y * z).
```

Here is an example of a formula, which is not (yet) a clause:

```
all x all y exists z ( x * z = y ).
```

Periods

Warning: all input formulas must end with a period (a “full stop” in British English).

(Don’t worry, you will forget this many times.)

Variables

Convention: Lower case letters at the end of the alphabet

```
x, y, z, u, w, v5, v6, v7, ...
```

are variables.

The letter “v” may be used in input files, but will not be used in output. This is because many people use it for the join operation in lattice theory, and so wish to avoid expressions like “v v v = v”.

Quantification

By default, variables are universally quantified if not otherwise indicated.

So

```
x * y = y * x.
```

means the same thing as

```
all x all y ( x * y = y * x ).
```

Constants

Lower case letters at the *beginning* of the alphabet and *all* upper case letters are constants, unless explicitly quantified.

So in the clause

```
c * x = x * c.
```

c is a constant that commutes with everything.

In the clause

```
all c ( c * x = x * c ).
```

c is a variable, and we have asserted that * is a commutative operation.

Clausification

Prover9's main inference process works with clauses only, not with more general formula. Thus in preprocessing, Prover9 *clausifies* all formulas.

For example, the cancellation law

```
x * y = x * z -> y = z.
```

will be clausified like this:

```
x * y != x * z | y = z.
```

Clausifying connectives

During preprocessing, logical connectives are re-expressed in a normal form.

For instance, negation is not allowed to be applied to a disjunction or a conjunction. So `-(a & b)` is written as `-a | -b`.

Example

I have been working on a loop theory problem that has the following nasty condition:

```
(x * y) * z = x * (y * z) & (x * z) * y = x * (z * y) |
(x * y) * z = x * (z * y) & (x * z) * y = x * (y * z) |
(x * y) * z = (x * z) * y & x * (y * z) = x * (z * y).
```

Prover9 distributes “or” over “and” and clausifies this to eight clauses

```
(x * y) * z = x * (y * z) | (x * y) * z = x * (z * y) | (x * y) * z = (x * z) * y.
(x * y) * z = x * (y * z) | (x * y) * z = x * (z * y) | x * (y * z) = x * (z * y).
```

etc.

Skolemization

If a formula contains an existence statement, Prover9 (and Mace4) will *skolemize* it during clausification.
This means it replaces existence statements with functions.
For instance, the formula

```
all x all y exists z ( x * z = y ).
```

asserts that z is a function of x and y. So the formula is skolemized to

```
x * f1(x,y) = y.
```

(Skolem functions are numbered `f1`, `f2`, etc.)

Denials

Prover9 proves everything by contradiction. Thus it takes whatever conjecture is in the goals list and forms its *denial* using constants.
For instance, the goal

```
x * y = y * x.
```

will be converted into the denial.

```
c1 * c2 != c2 * c1.
```

Multiple goals are allowed, but as a rule of thumb: don't.

Key Ideas

There are three key ideas in Prover9's inference process: the *given clause*, the *usable list* and the *sos* (set of support) *list*.

The sos list is the list of clauses that are waiting to be selected as given clauses. Clauses in the sos list are not available for making primary inferences, but they can be used to simplify inferred clauses by rewriting and unit deletion.

The usable list is the list of clauses that are available for making inferences with the given clause

At each iteration of the loop, a given clause is selected from the sos list, moved to the usable list, and then inferences are made using the given clause and other clauses in the usable list.

Precise description

While the sos list is not empty:

1. Select a given clause from sos and move it to the usable list
2. Infer new clauses using the inference rules in effect; each new clause must have the given clause as one of its parents and members of the usable list as its other parents;
3. process each new clause;
4. append new clauses that pass the retention tests to the sos list.

end of while loop.

Paramodulation

For equational problems, the most important inference rule is *paramodulation*. This is equational manipulation at its most basic, substitution of terms into equations.

To see how it works, let's look at a small piece of a Prover9 proof:

```
23 (x ^ ((y ^ x) v z)) ^ (y ^ x) = y ^ x.
130 ((x ^ y) ^ z) v x = x.
346 (x ^ y) ^ ((y ^ z) ^ x) = (y ^ z) ^ x.
                [para(130(a,1),23(a,1,1,2))].
```

(Let's work this out on the board.)

Binary Resolution

The basic idea of resolution is that if one of the literals in a disjunction is falsified, then the remaining literals are true. Here is a simple example:

```
18 x * y != z | z * y' = x.
20 ((x * y) * z) * y = x * ((y * z) * y).
(x * ((y * z) * y)) * y' = (x * y) * z.
                    [resolve(18,a,20,a)].
```

(Let's work this out on the board.)

Hyper-resolution

Hyper-resolution is an inference rule that allows multiple binary resolutions all at once: Example:

```
2 -P(e(x,y)) | -P(x) | P(y).
3 P(e(x,e(x,e(y,z)))).
5 P(e(e(x,e(x,e(y,z))),e(u,w))).
          [hyper(2,a,3,a,b,3,a)].
```

(Let's work this out on the board.)

Negative hyper-resolution

Negative hyper-resolution works “backwards”.
Example:

```
5 x * y != z * y | x = z.
14 (c1 * c2) * c3 != c1 * (c2 * c3).
((c1 * c2) * c3) * x != (c1 * (c2 * c3)) * x.
[hyper(5,b,14,a)].
```

UR-Resolution

The *UR-resolution* (unit resolving) rule can be thought of as a jazzed up version of hyper-resolution which mixes positive and negative clauses. In problems with clauses with few literals, it behaves much like hyper-resolution.

Negative UR-resolution is often turned on by default. It is frequently desirable to turn it off as we will discuss later.

Rewriting

In equational problems, an important task of provers is *rewriting* (or *demodulation*) by replacing terms by something "simpler". The definition of "simpler" depends on the choice of term ordering.
For instance, say that Prover9 derives an equation such as

```
x * (y * z) = x * z.
```

This will now play the role of a *demodulator*. Perhaps in the sos, there is some equation like

```
..... = x * (y * z).
```

Prover9 will immediately rewrite it as

```
..... = x * z.
```

using the demodulator.

Function order

During preprocessing, Prover9 finds all of the constants, functions, operations, etc. in the input file and places them in a certain order:

```
function_order([0,1,c1,c2,f,g,*,/,\]).
```

Constants come first (including constants from denials), followed by unary functions, binary functions, etc.

If the results of the default `function_order` are unsatisfactory, the user has the option of specifying it.

Term ordering

Prover9 has two main term ordering options, which tell it how to decide to rewrite terms:

LPO - lexicographic path ordering. This tells Prover9 to rewrite terms by following `function_order` whenever possible.

KBO - Knuth-Bendix ordering. Roughly speaking, this tells Prover9 to rewrite terms in the shortest way possible, using `function_order` if there are multiple options.

LPO is the default, because in some problems, it is the faster option.

Difficulties with LPO

LPO can sometimes cause terms to “blow up”. For instance, suppose we have

```
function_order([E,*,\]).
```

and suppose that Prover9 derives the clause

```
x \ y = E * (x * (E * y)).
```

LPO will cause Prover9 to rewrite all instances of in terms of this new demodulator. This can significantly increase the sizes of clauses.

Equational definitions

Say you have an equational definition, like this:

```
L(x,y,z) = (x * y) \ (x * (y * z)).
```

By default, during preprocessing, Prover9 will expand all other input instances of `L(x,y,z)` in terms of the other operations, and it will never be seen again.

But sometimes you *want* Prover9 to keep the function and to rewrite using the function whenever possible. In that case, you can force Prover9 to move the definition up front in `function_order` using this option:

```
set(eq_defs).
```

Simplification

After a clause is inferred, many things, happen to it. First, it is simplified. Say that C is the newly inferred clause.

- Rewrite C using all available demodulators.
- Orient the equalities, heavy to light.
  For instance, `x * y = z` is oriented, `z = x * y` is not.
- If C contains any identical literals, merge them.
  For instance, the clause
  `x * y = y * x | x * y = y * x`
  will be merged to just `x * y = y * x`.

Simplification II

- Delete units in *C*.
  For instance, suppose that after simplification, *C* contains `x != x | x * y = y * x`. The obviously false literal will be deleted, leaving just `x * y = y * x`.
- *CAC redundancy* check: if there is a commutative or commutative-associative operation, and if *C* contains an equality which is a consequence of this property, then the equality will be simplified to TRUE and deleted.

Limit checks

After Prover9 has finished simplifying our clause C, it checks to see if C passes certain limits. These are

`max_weight` `max_literals` `max_depth` `max_vars`

Max weight

The *weight* of a clause is based on counting symbols. In equational problems, the equality symbol is counted, but not parentheses.
For instance, the associative law

```
(x * y) * z = x * (y * z).
```

has weight 11.

The default `max_weight` is 100. If a clause has weight more than this, it will be deleted.
For many problems, the default is much higher than necessary. Lowering it can significantly reduce the search space.

Max literals

By default, Prover9 has no limit on the number of literals a clause may have. Thus the value of `max_literals` is -1.

If your input file has clauses with more than one literal, like this:

```
(x * y) * z = x * (y * z) | (x * y) * z = x * (z * y).
```

then Prover9 might derive clauses with even more literals. If you suspect that such clauses will not help, try reducing the value of the flag `max_literals`.

Max variables

By default, Prover9 has no limit on the number of variables a clause may have. Thus the value of `max_vars` is -1.

If you have reasons for thinking that a proof should not require too many variables, one way to reduce the search space is to reduce the value of the flag `max_vars`.

Max depth

Informally, the *depth* of a clause measures how deeply nested expressions are.

For example, `-p(f(x))` has depth 2.

By default, Prover9 has no limit on the depth of a clause may have. Thus the value of `max_depth` is -1.

Max depth 2

Sometimes, we want to avoid clauses that “overevaluate” functions.
For instance, if we have a function `f(x)` in our input, we might suspect that clauses of the form

```
f(f(f(f(f(f(f(x)))))))  =  x.
```

will not be useful. Reducing the value of `max_depth` might help.

sos limit

If our poor clause *C* has gotten this far, Prover9 now checks to see how full the sos list is. The default value of `sos_limit` is 20000. As the size of the sos approaches that number, Prover9 starts discarding "worst" clauses from the sos.

"Worst" is roughly based on weight. The "worst" clause that we still have will be deleted to make room for *C* (unless *C* is even worse).

Forward subsumption

Assuming C still survives, now Prover9 checks to see it is subsumed by some previously derived clause, say B.

Clause B *subsumes* clause C if the variables of B can be instantiated in such a way that it becomes a subclause of C.

Kept!

Finally, if *C* has gotten past all that, it is assigned a clause ID number and *kept*.

And *now*, Prover9 checks to see if *C* gives a *unit conflict*, that is, does *C* contradict some other kept clause. If so, the search is finished and we have a proof.

This is called *unsafe* unit conflict. What can happen is that *C* might be a clause that will finish the search, but it gets deleted by some limit (say, `max_weight`) before it is checked.

There is an option for *safe* unit conflict checking, before limits are applied, but it is computationally expensive.

Using the new clause

Assuming the search is not over, *C* now gets used in various ways.

For instance, Prover9 checks to see if *C back subsumes* already derived clauses. This is expensive, though, and Prover9 will stop doing it after 500 given clauses (that value can be changed).

If *C* can be used as a demodulator, Prover9 will use it to rewrite clauses in the sos.

Finally, after a couple of more things I will skip, Prover9 moves *C* to the sos list.

Given clause selection

How does Prover9 choose which clause in sos should be the given clause?

There are many strategies for this.
A *breadth first* search simply takes the oldest available clause.
A *lightest first* search takes the lightest available clause.

In practice, the best strategy is a mix: select an old clause, then a few light clauses, then an old clause, etc.

Given clause selection 2

Prover9's default selection scheme works in a cycle of 9 steps:

- Select the oldest available clause
- Select the four lightest available "false" clauses
- Select the four lightest available "true" clauses

For now, you can think of "false" as negative and "true" as nonnegative.

For instance, the denial of the goal is a negative clause. If Prover9 reasons backwards from the denial, this will generate new clauses.

Later, when we discuss semantic guidance, we will expand the definition of "true" and "false".

Given clause selection 3

The values “1 old, 4 true, 4 false” were chosen experimentally.
In many problems, there are few false clauses available, so it effectively becomes “1 old, 4 true”.

The numbers can be easily changed.
Later we will discuss how to modify the given selection scheme for specialized purposes.

“It’s all about the given clause.” – Bill McCune

Examples

Let's now work through some examples illustrating various ideas.