AUTOMATED DEDUCTION AND ALGEBRA
Lecture III

Michael K. Kinyon

Department of Mathematics

UNIVERSITY OF DENVER

University of Lisbon, 15 June 2009

Tweaking the Cycle

The default “1 old, 4 light true, 4 light false” given clause selection cycle was chosen by the developer because in a series of experiments with some standard examples, it gave proofs faster than other possibilities.

But that does not mean it is good for the problem you have at hand. Fortunately, it is easily modifiable.

Tweaking the Cycle II

The easiest way to modify the cycle is to change the values assigned to its *parts*.

A more sophisticated way involves writing one's own cycle:

```
list(given_selection)
part(Hint,high,weight,hint ) = 1.
part(Age, low, age,    all  ) = 1.
part(True,low, weight,true) = 4.
part(Weird,low,random,false & vars > 3) = 2.
```

Many rules for choosing a clause can be put in the fourth slot.

Plans

The reason for occasionally picking an “old” clause is that if we choose clauses lightest first, something important may be missed.

An idea I am currently working on is to try to find a better default cycle with schemes like this:

```
list(given_selection)
part(H,high,weight,hint ) = 1.
part(W1, low, age, 0 < weight & weight <= 10) = 10.
part(W2, low, age, 10 < weight & weight <= 20) = 8.
part(W3, low, age, 20 < weight & weight <= 30) = 6.
etc.
```

This will require a lot of experiments.

Deduction Over Finite Structures

Although in principle it is possible to specify finite domains for Prover9 and other provers, they are generally not good at working with such structures.

Here is an example that actually works, but the complexity of the proof indicates what the problem is for larger structures.

Weighting

Weighting is a method of assigning specific weights to clauses based on what terms they contain.

```
list(weights).
  weight(a) = 3.
  weight(f(a,x)) = 5 * weight(x).
  weight(f(a,_)) = -1.
  weight(x | y) = 2 + (weight(x) + weight(y)).
end_of_list.
```

Semantic Guidance

Next we will turn to another technique for influencing a search called *semantic guidance*. The idea is based on the following somewhat trivial observation:

Some input formulas are more important than others.

So we would like Prover9 to pay more attention to those formulas and their children than the rest.

One way to do this is to reserve part of the given clause selection cycle for the important clauses. But how do we mark them as important?

Interpretations

We use Mace4 to generate models in which the less important clauses are true, but the more important clauses are false.

We put the models into an *interpretations list*.

When a clause is kept, Prover9 evaluates the clause in the models. If the clause is false, that is, if it fails to hold in any model, then the clause is labeled “false”.

False derived clauses *must* be children of the false input clauses!

Interpretations II

When it is time for a False clause to be selected as given, Prover9 selects the lightest clause labeled “false”.

When it is time for a True clause to be selected as given, Prover9 selects the lightest clause *not* labeled “false” (which presumably means the clause is true).

Limitations

- Evaluation of a clause in a model can be expensive if the model is large or if the number of variables is high. Thus Prover9 has (modifiable) restrictions on what it will actually evaluate. For instance, in a model of size 8, it will only evaluate clauses with at most 2 variables. Clauses left unevaluated are treated as “true”.
- For the same reason, for large jobs, one can quickly run into memory restrictions.

The Default Interpretation

If the interpretations list is empty, then Prover9 uses the *default interpretation*:

- Negative clauses (*e.g.*, children of denials) are false
- Nonnegative clauses are true.

Now you know why parts of the given clause cycle are called "False", even though strictly speaking, the clauses aren't false.

Quasivarieties and Varieties

The same idea that underlies semantic guidance can actually help us *discover* new results as well.

Suppose, for instance, that we have some identities and a quasi-identity in the input. Then the structure we are studying is a quasivariety (closed under substructures and products, but possibly not homomorphic images). Perhaps it is a variety?

Ugly Proofs

Provers like Prover9 rarely (= never) find “optimal” proofs, they just find the first available proof. Sometimes this does not matter, since we may have wanted only a yes-or-no answer to the question of whether something is true.

But sometimes we may want to translate a proof into human form. Or perhaps a journal editor will allow us to publish the Prover9 proof, but only if it is not too long.

So what can we do when we get a proof that is a real monster?

Naive measures

Here are some criteria for measuring simplicity of a proof:

- *Length*. (But be careful: Prover9's "Length of Proof" does not count rewrites.)
- *Level*. Shallower proofs are easier to follow.
- *Max clause weight*. Usually a proof with smaller clauses is easier to follow.

Use hints!

The basic idea is to rerun the Prover9 job using the first proof as hints.

This usually (but not always) results in a simpler proof, as measured naively. This process can be repeated several times. Sometimes the length and depth will go up. If they don't go up too much, that's OK; they might go down in the next iteration.

Control the max weight

To keep the new search space reasonable, we lower the maximum clause weight to be *equal* to that of our first proof.

By default, the maximum weight, variables, literals and depth limits do not apply to hint matchers. However, this can be modified as follows:

```
set(limit_hint_matchers).
assign(max_weight, ...).
```

Start with a stable proof

Call a proof *stable* if, when the proof is fed back to Prover9 as hints, Prover9 generates precisely the same proof.

The first proof found is almost never stable. The preceding process of running and rerunning jobs often leads to a stable proof.

Having a stable proof is useful as a starting place from which to try simplification.

Squeeze the max weight

We set `max_weight` to be 1 less than the maximum clause weight of the last stable proof we found, and start again.

Assuming a new proof is found, it is might be more complex in length and depth than the last stable proof or less complex or about the same.

In either case, we again set the maximum weight *equal* to the proof's max clause weight and stabilize the proof.

Save often!

This is a lot like playing a video game: we need to save proofs quite often. Sometimes a proof we find midway through this process will be much simpler than any proof found later.

Change how hints are selected

Eventually, the maximum weight will be as low as it can be for Prover9 to find a proof; setting it any lower will lead to a failed search.

By default, hint matchers are chosen lightest first. This can be changed:

```
set(breadth_first_hints).
```

This can “shake up” the search space enough to allow a new proof to be found. This can be turned on and off repeatedly just to see what happens.

Play with given selection

Assuming the most recent proof is stable, another technique to simplify the proof is to modify the given clause cycle.

By default, hint matchers have high priority. We can change this to force Prover9 to *avoid* hint matchers and to choose other clauses for a while.

Other techniques

- *Term weighting.* Pesky clauses can be “blocked” from participating in a proof by putting them in a weight list and setting their weights larger than the maximum clause weight. (This usually only leads to minor improvements.)
- *Combine hints from different proofs.*
- *Start over.* Start again with no hints and a small max weight and see if that helps.

Most importantly

Stop!

Don't waste hours on this. There is a law of diminishing returns.

If you can remove several hundred steps from a proof, go ahead.

But if you spend hours shortening a proof by one step, you should go outside and get some fresh air.