

CSCI8980, Deduction and Computation, Fall 2004

Optional Programming Problems

Posted: October 31, 2004

Some of you expressed an interest in class in understanding the kinds of programming problems that could be relevant to what we have discussed in class. Here are some examples of things that you could try. Hopefully you will be able to see practical connections through some of this as well: for example, an interactive theorem-prover can have the role of a sophisticated calculator and there are lots of situations (such as expert systems or advise systems) where simple theorems have to be proved and where interface issues (such as translation of a proof into natural language) become important. However, none of these problems are about an explicit application. I don't see this as a Bad Thing, though, since our interest in this course is to abstract away from that situation to have something with enough structure and long-term interest to study.

You are not required to solve any of these problems. However, even understanding them will be useful and, if you end up working on any one of them, I would be happy to talk to you and to help you along.

Problem 1

We talked about resolution and the sequent calculus as means for showing that a propositional formula is tautologous and I mentioned writing a program to realize either of these systems. Here is a variant that underlies what is known as the mating method in automated theorem proving and that you might find more interesting because it is different.

First, we visualize a formula in negation normal form as a matrix arrangement of literals. In this arrangement, disjunctions give rise to rows and conjunctions lead to columns. Thus, if A , B and C are literals, then the formula

$$(A \wedge (B \vee C)) \vee (\neg A \wedge B) \vee \neg B$$

can be visualized as the matrix

$$\left[\begin{array}{c} \left[\begin{array}{c} A \quad \left[\begin{array}{c} B \\ C \end{array} \right] \end{array} \right] \\ \left[\begin{array}{c} \neg A \quad B \\ \neg B \end{array} \right] \end{array} \right]$$

Now, in such an arrangement, you can think of a *path* as a sequence of literals that is obtained by moving from the top of the matrix to the bottom picking up a literal at each stop, so to speak. Thus, the sequence $[A, \neg A, \neg B]$ is one path and $[B, C, \neg A, \neg B]$ is another. You can easily construct an argument to the effect that a formula is tautologous if and only if all the paths corresponding to it have a pair of complementary literals; we call a path that has this property *closed* below.

You can write a program for showing tautologousness that utilizes this insight. Here may be the steps to carry out:

1. Write a program to convert any given formula into negation normal form.
2. Now write a program to take the negation normal form and transform it into a matrix representation.
3. Write a program that takes this matrix and shows that all the paths in it are closed. In the simplest form, this program would first generate all the paths and then check each one for closedness. However, there is a lot of scope for exploring heuristics for cutting down the work in this context.

In developing this program, you will have to think of representations of formulas, matrices, etc. I have left all these details implicit in this description.

Problem 2

We are discussing searching for sequent proofs in first-order logic in class. You can experiment with various versions of the ideas we discuss at a programming level. Here are a few possibilities:

1. One interesting version is to build a framework for constructing proofs interactively. The simplest step we might think of applying in this process is an inference rule. In the programming version, we might call this a *tactic* that takes the end sequent as input and that returns the upper sequent(s) as new goal(s) in the proof search process. Now, you may want to combine these steps into larger ones in meaningful ways. For instance, you may want to generate a derived rule that says that if you have a formula of the form $(\forall x_1) \dots (\forall x_n)(G \supset A)$ on the left of the sequent and you have a formula of the form A' on the right, then one way to proceed would be to replace all the quantified variables by new (instantiatable variables, see if you can unify A and A' and then try to show G follows from the same premises. If you work through this, you will see that this tactic, that we might call *backchain* is really a combination of \forall -L*, \supset -L and an axiom. Thus, you can write a function called a *tactical*, that combines these steps to produce the *backchain* tactic. Note that the user can use the such tacticals to construct interesting tactics “on-the-fly,” as the proof construction process demands it. In a different direction, you can also develop a “giant” tactic that runs one complete round of rule applications other than \forall -L* and \exists -R* and returns new goal sequents to which we may have to apply the omitted quantifier rules to proceed further. Once you have this system, the user would provide the formula and then direct the construction of new tactics, their application to a given formula, etc, in the direction of finding a proof.
2. You could build a completely automatic system. This may follow the control regimen that we are discussing in class: Try all rules except \forall -L* and \exists -R* till you reach a sequent to which nothing else can be applied. At this stage think of using the \forall -L* and \exists -R* if possible and repeat the process one level deeper, so to speak. You will have to think of how to deal with the eigenvariable constraint in these situations. We have talked of one approach in class but there can be more efficient ones. Ask me and I will give you some ideas.

3. In both approaches discussed till now, the idea has been only to trace out the rule application process in the execution structure of the program and not to return a proof *per se*. However, quite often one wants to return a proof. Think of augmenting your program so that a proof representation is returned.

Writing the programs suggested here will require you to do some or all of the following: develop a unification procedure, implement substitution into quantified formulas correctly and develop representations of quantified formulas and also for proofs.

Problem 3

You can eliminate essential universal quantifiers from first-order formulas in the classical context using a static “Herbrandization” process as we shall see in class. In this situation, you can think of lifting the matrix approach to theorem proving discussed in the first problem to the first-order situation in the following way: first generate a matrix by instantiating each existential quantifier with a new variable and then try to show that all the paths in the resulting matrix is closed; you can bind the new variables to specific terms in this process. If this fails, generate two copies of the part of the matrix governed by some existential quantifier and then try again. Fill out the details of the approach (that is applying what is called an *amplification* to the quantified parts) that I have sketched out, understand why it is complete based on the discussions in class and try to implement a theorem-proving procedure based on it.

Problem 4

The kind of reasoning that we are looking at in class is useful in a number of situations. People have tried to automate it to build systems that prove interesting new theorems in, say, mathematics. However, there are also possible uses in situations where the theorems are themselves not very deep but they are still useful in, for instance, giving people advice in practical situations. Of course, this advice should be played back in understandable language for it to be useful for the kind of user we are thinking about. This brings up an interest in trying to generate a natural language rendition of a proof of something manifest in a sequent calculus or natural deduction form. Write a program to do this. You can assume as input a proof (that a system like the one in Problem 2 or 3 has found) in a natural deduction calculus, for instance, and the job is to translate this into one in English. Of course, you will have to develop a representation for a natural deduction proof first.

Problem 5

Transformations of proofs can be of interest in a number of different contexts. Here are a few possibilities that you can try your hand on:

1. Read the proof of cut-elimination in Gentzen’s paper and try to encode its insights into a program that takes a proof possibly with cuts and transforms it into one without cuts. This kind of an exercise has been viewed as a particular challenge for proposals for type systems for programming languages recently. For example, is there a way to encode proofs or to enrich the type system so that the program that you write can be guaranteed to return a cut-free proof? If you can do this, you are getting

partial correctness from the type system in this case. Even if you do not care for cut-elimination, there can be ideas here that you may find fascinating and useful in real, practical programming languages.

2. Suppose you have succeeded in finding a proof for a formula using the matrix method, can you transform this information into a natural deduction or sequent proof that can be presented to the user in an understandable way? The solution to this problem can be found by extrapolating from things that we will see in class, but it also could be more challenging than is fair. So think a little about it and then ask me for a reference that will take you through the answer. Once you have the solution, you can write a program to implement it.