TDDD08: Some Exam Remarks

Victor Lagerkvist

This document contains some tips which will (hopefully) make the study period a bit smoother. It should not in any way be viewed as an indication for which problems might or might not appear during the exam.

Why are there no solutions to old exams?

Studying for an exam by learning from old solutions can easily become a form of rote learning where one ignores the bigger picture and only concentrates on memorising solutions. After graduation you will practially never have the opportunity to compare your own solution to a given solution. In fact, in general there are no perfect solutions, only approximations, and thinking that you cannot solve a problem without comparing it to a pre-given solution can lead to very bad habits of mind. For most types of exercises in this course there is not one, but typically many, good solutions, and you cannot in general verify whether your own solution is correct by comparing it to a pregiven solution. Note that TDDD08 is a course on *advanced* level and that by this stage in your education you should be able to prepare for an exam without basing it solely on old solutions.

Last, there is a *wealth* of exercises and detailed solutions in the course book, the lecture slides, and the tutorial exercises. And you can easily find more examples online if you so desire. Hence, if you think that it is easier to learn a certain technique (e.g. learning to draw SLDNF-forests) by solving exercises with detailed solutions then you already have the possibility of doing so. Importantly, there is nothing magical about exam exercises which makes them stand out from other exercises which you have encountered in the course (a logic program is still a logic program, an SLD-tree is still an SLD-tree, and so on).

Thus:

1. Focus on tutorials, exercises from the course book, and the lecture slides/videos for learning a technique or a method.
2. Solve exam exercises to hone your skills and look through old exams to get a better sense of what is expected, knowledge-wise. Here, it is also a good idea to check which assumptions you are allowed to make during exam-like exercises. For example, unless specified otherwise, you are not allowed to use non-declarative features of Prolog such as the cut.

3. Ask us if you are unsure of what an acceptable exam answer would be in a certain scenario, e.g., what the write-up should consist of. However, keep in mind that the answer is typically very straightforward: your solution should include everything, including all step-by-step calculations that you have done, which allows us to verify your solution. And if you write a program it is a very good idea to clearly comment each line, and to informally describe the intended relation of each predicate. This makes the grading easier and increases the odds of getting credits for a solution even if it might not be formally entirely correct.

Why do I have to program on pen-and-paper during the exam?

Most exams feature one or two programming exercises where you are asked to produce a logic or Prolog program solving a given task. Here, we wish to test your declarative problem solving ability, and we are not interested in whether you fully recall the syntax of Prolog or not, or if you have made silly errors that would have been detected immediately during an actual implementation of the program. In fact, asking you to produce a program using nothing else than your wits and pen-and-paper actually makes the task easier for you since you can concentrate fully on solving the problem, rather than getting distracted by tedious programming errors.

How can I verify my own solutions?

This is in general a difficult problem without a perfect solution. Thankfully it is substantially easier for many of the techniques and algorithms which we have developed during the course (otherwise, we as teachers would find it very difficulty to grade your solutions).

Example 1. *Assume that you are asked to produce a program solving a given task, e.g., defining a predicate for a given relation. First, try to understand the intended relation(s). It may help to make some examples: a few representative ground atoms which should be answers of your program (in other words, queries where Prolog should answer "yes"). Similarly, construct a few ground atoms which should not be answers. Do not forget border cases (like an empty list as an argument, an atom similar to a required answer, and so on).*

Convince yourself that your program will produce the required answers and will not produce the non-required ones. This may be done as follows.

While writing your program, take care that each atom which should be an answer is produced by some clause of the program (out of some other atoms that should be answers). This means that the atom has to be an instance of *a fact, or of the head of a rule from the program. Moreover, in the latter case the body atoms of the rule instance should also be required to be answers.*

On the other hand, no clause of your program is allowed to produce an atom that should not be its answer (out of atoms that may be answers). So check each clause for this: such an atom cannot be an instance of a fact. If such an atom is the head of an instance of a rule, then at least one of the body atoms should not be producible by the program.

Performing the above steps should not take more than a couple of minutes, and if you do not succeed this could be a sign that you either have forgotten something in your program, or that you do not fully understand your own program. During the study period, you could also test your program in a Prolog system. While tests of this form are of course not a substitute for full correctness and completeness proofs, they greatly increase the chance of finding a correct solution.

Note that if you are not able to produce any test cases, there there is a large risk that you do not understand the given assignment. In this case it is typically much better to re-read the assignment once more, possibly trying to look up any concepts that are difficult to understand, than trying to produce a correct program.

Example 2. *Assume that you are asked to verify whether two given terms are unifiable or not. Note that it is typically not so easy to "see" whether the terms are unifiable, and intuition can in many cases mislead us, so it is much better to attempt to follow the unification algorithm from the lecture slides. If you are then uncertain whether your solution is correct, simply try to (1) apply your MGU to the two terms and check that the resulting two terms are indeed equal, or (2) double check all steps leading to a reject answer in the unification algorithm. If you are practicing for the exam then you can also simply try to unify the two terms in Prolog and see what you get (be aware of the fact that Prolog omits the occurs check, and consider using* `unify_with_occurs_check/2` *instead of* `=/2`*).*

Example 3. *A common type of exercise is to produce the set of atomic logical consequences or the set of ground atomic logical consequences of a given logic program (i.e., the least Herbrand model). Try to follow the bottom-up method for constructing the least Herbrand model, as done during the lecture and the corresponding tutorial. Once you have a set of ground atoms which you believe might be a correct answer, pick a few (not so big) ground terms and see whether they are provable from the program or not, e.g., by drawing the corresponding proof trees. If you are practicing before the exam you may also test out the program and a few queries in Prolog. Be aware of the possibility of infinite loops in Prolog due to Prolog's search strategy (and consider using* `call_with_depth_limit/3` *in SWI-Prolog to circumvent this).*

The above hints are applicable to most exercises where you are asked to *produce*, or verify, something, e.g., producing a program, a DCG, or checking whether two terms are unifiable. Another class of exercises is where you are asked to *prove* something. This is, in general, harder to verify, but when we ask you to prove something it is typically specified how it should be accomplished. The most frequent types of exercises of this kind are proofs of correctness or completeness of programs. Here, it is not so easy to "test" if a proof is correct or not, but before attempting to produce a proof it is a good idea to try to understand the given program. If you are practicing before the exam, why not test a few example queries in Prolog? Do you obtain anything that is unexpected? It can be useful to have this in mind once you proceed to the actual proof since you will then have a better intuition why or why not a program is correct with respect to a given specification, or not.

How do I know that my solution is sufficently detailed? What do I need to include?

This problem is related to the previous one but still a bit different in nature. We already have detailed solutions to similar exercises in the course book, during the lectures, and in the tutorials. If your level of detail is very different to those suggestions then it is very likely not sufficient. In particular, if you have a question along the lines of "Do I really need to include this part?" then the answer is very likely affirmative. We *never* deduct points for superflous details, but we *always* have to deduct points when important details are missing.

What if I do not even understand the exercise?

Then you will have to go back to the original material and study the basic concepts (e.g., tutorial exercises, lectures, or the course book. Try to find the most fundamental concept which you do not understand and concentrate on that. Yes, this can sometimes be more time consuming, but you will also pick up other skills in the process, increasing the likelihood of solving related (but not identical) exercises.