CSE505 HW1 Grading Information

Andy Collins

October 22, 2003

My goal in writing this is to tell you how the homework went overall, to help in interpreting your marks, and to discuss some common errors and issues that came up in grading the assignments. Obviously you should direct individualized questions about grading to me personally.

Overall, I thought most people did pretty well, although as I discuss more below, many of the proofs could be a lot tighter and crisper. I went easy on it this time, but I'd like to tighten up on that as the quarter progresses.

1 Point breakdown

But first, some purely mechanical stuff. Homework 1 was graded out of 30 points, broken down as follows:

- **Problem 0:** 5 points (1 for the basic tree stuff, 2 for `least_greater`, and 2 for the use of polymorphism). As a rule I assumed that if your int trees worked correctly so did your polymorphic trees; the points for the latter were all for the polymorphism itself.
- **Problem 1:** 4 points (2 for (a), 1 each for (b) and (c)).
- **Problem 2:** 9 points. 1 point each for (a) and (b). 1 point each for (c.i), (c.ii), and (c.iv). 2 points for (c.iii) (which had an actual proof, rather than just a counter-example). And 2 points for (d), which I graded primarily by running test-cases.
- **Problem 3:** 5 points (2 points each for (a) and (b), and 1 point for (c)).
- **Problem 4:** 7 points. 1 point each for (a), (c), (d.i), and (d.ii). 3 points for (b). I graded (a) mostly by inspection, although some of the testcases also pointed out problems. I graded (b) mostly by running testcases.

For those keeping count, this breaks down to 16 points for the written parts, 9 for the interpreter code, and 5 for the tree code.

1.1 Reading my notes

If you can't read my writing, then I apologize, and you are welcome to stop by and make me try to interpret it for you. I can usually read my own writing about nine times out of ten.

For the most part, and knowing that being UW students everyone would be closer to perfect than to zero, I chose to mark the papers using negative grading, marking off points and then subtracting those values from 30. Sometimes I wrote "1/2" for one point out of two on a part. When marking half points (which I try to avoid but couldn't keep myself from doing sometimes), I always marked ".5" so fractions are always points out-of.

1.2 Don't panic

I ran everyone's code through a very rough script to compile and test, and it failed for reasons not your fault way more often than I would really have liked. Therefore, many of you will see log files in the printed code I've attached that make it look like your code choked and bombed when in fact it ran fine on the second pass. This is why the handwritten notes often seem to disagree with the printed logs. Nobody lost points because my script couldn't make their code run.

2 Interpreter testcases

I used three main and two ancillary IMP programs to test your interpreters and `prevent_error` analyses. I've put all five files on the webpage. I used them as follows:

- `test-interp1.imp` is a general test that should run under either version of the interpreter. It tests both ntimes and code pointers. The expected output is `5555`. It contains a violation of `prevent_error` (a run of something that was once an int), but no actual runtime error. It tests both a "conventional" ntimes and the vexing "modify-the-expression" ntimes. I called it "testcase 1" in my notes.
- `test-prevent2.imp` is a simple test of `prevent_error` in the case of dead code. It does not contain a runtime error, and the expected output is `5`. I called it "testcase 2" in my notes.
- `test-prevent3.imp` is a more interesting case of a static (and also a runtime) error. This tripped up many people who forgot to check the statement part of an AssignStmt as part of `prevent_error`. I called it "testcase 3" in my notes.
- `test-noerror.imp` is the only testcase that uses Run but doesn't have a static error. I only ran this manually when I suspected I was mis-interpreting something.
- `test-prevent4.imp` is something I cooked up after I did the main run, to catch implementations of `prevent_error` that weren't proper static analyses, and were instead trying to actually run the program and get perfect error information. Although the program itself is an infinite loop, it is still legal to run `prevent_error` on it, and doing so should still terminate.

3 Notes on the questions

3.1 Question 0: O'Caml Warm-Up

People mostly did pretty well here. On a completely non-505-related note, a lot of people were sloppy about documenting what happened if the same item was inserted multiple times, although this is less of an issue since we didn't implement remove. I also *strongly* recommend people take a look at the sample solution, even if they got full marks, because many of you made the solution a lot more complex than it needed to be. Languages like O'Caml admit really elegant solutions to recursive problems, and trying to fit an imperitive model to them gets really ugly in a hurry. Nobody lost points for ugliness, but that doesn't mean it isn't worth learning how to write elegant solutions.

3.2 Question 1: Interpreter Warm-Up

Most everybody did fine here.

3.3 Question 2: ntimes

The main error here (and this was unsurprisingly reflected in both the definition of the operational semantics and the interpreter code), was re-evaluating the expression after the statement had executed one or more times.

There were, in fact, two workable solutions to part (a) that used only a single rule. The first is essentially similar to the sample solution, but instead of separating the base case from the recursive case at the rule-selection level, it did so using the IMP if construct:

$$\frac{H \ ; \ e \ \Downarrow \ c}{H \ ; \ \mathsf{ntimes}\ e\ (s) \ \rightarrow \ H \ ; \ \mathsf{if}\ (c)\ (s; \mathsf{ntimes}\ (c-1)\ (s))\ \mathit{skip}}$$

The critical aspect to making this work is requiring the evaluation of e to the constant c. We end up constructing another expression $c-1$ that will be evaluated later on, but the value of that expression can never change. The much more subtle variation looked like:

$$\frac{}{H \ ; \ \mathsf{ntimes}\ e\ (s) \ \rightarrow \ H \ ; \ \mathsf{if}\ (e)\ (\mathsf{ntimes}\ (e + (-1))\ (s); s)\ \mathit{skip}}$$

This actually requires nothing (i.e. it is an axiom). It constructs a new expression based on the given expression e (note the use of the IMP plus operator, and the constant -1, because IMP has no minus). The expression e is in fact evaluated multiple times here, but because we expand the statement left-recursively (i.e. we put the unrolled statement *after* the recursively-expanded part) we know that it won't change between evaluations.

I want to emphasise that, although these are both acceptable solutions, and they both get by with just one rule, I do not believe that either of them is "better" or more elegant than the two-rule sample solution. This is, of course, a matter of opinion, but I will offer that the *interpreters* based on these rules were typically trickier, more complex, and less efficient than those based on the two-rule solution, and even worse, the proofs presented for part (c.iii) were much uglier and more error-prone.

3.4 Question 2.c.iii: Proving termination of ntimes

In retrospect, this proof has a property that tends to encourage sloppy proof construction, and even the sample solution probably said less than it should have. At issue is the required strengthening of the inductive hypothesis. As we've seen several times in lecture, the typical model of a proof by induction involves taking the thing to be proven and transforming it into a stronger statement that will be our inductive hypothesis. Only after stating the inductive hypothesis and declaring the aspect on which we are doing the induction do we identify and prove the base case(s) and prove the inductive step.

In this case, no strengthening was required, and the inductive hypothesis was exactly the theorem to be proven. As a result, it was easy to get away without explicitly stating the inductive hypothesis, and this, in turn, made it easy to be unclear about exactly on which quantity we were performing the induction, and on which assumptions and results of the inductive hypothesis we were relying as we proved the inductive step. You should not count on this being true later on, and you should make sure that you are really comfortable with precisely specifying inductive proofs before you get too casual.

The most common area of sloppiness here was confusion about the quantity on which we were doing the induction. This is the value c to which the expression e evaluates. It is technically incorrect to talk about "induction on the expression e" because this is ambiguous, and could easily be interpreted as structural induction on the *complexity* of e, which is totally different. Talking about induction on "the number of times the loop executes" is also sloppy, because part of the point is to prove that this is properly bounded.

The other common area of sloppy exposition (although I'm sure most everybody understood it at some level), was the precise reliance on the assumption "for all H we know $H; s$ terminates" and the "for all H" around the entire theorem. Notice how the sample proof carefully shows that the H we start with is likely modified by the execution of the first unrolling, but that we still fit within the rules for applying the inductive hypothesis on the remaining iterations.