CSE 505 Assignment 2 Solution and Grading Guide

Andy Collins
acollins@cs.washington.edu

November 3, 2003

This is intended to be a combination sample solution, explanation of the sample solution, grading guide (mostly for me), and grading feedback document. We'll see if this works better than the arrangement from homework 1. One implication is that I will have to update this document after I'm done grading, to roll in feedback about how people actually did.

Point distribution

Like homework 1, homework 2 is graded out of 30 points. Overall, question 1 was worth 10 points, and question 2 (which had more parts and more proofs) 20. Question (2f) was worth 5, which seemed reasonable to me. Whether you think of question (2f) as extra credit is purely a matter of perspective; I chose to mark it as part of the 30 points possible, but if you want to think of it as true extra credit, then the assignment is out of 25 points. Breaking down the parts of the questions, (1a) and (1b) were worth 3 points each, so (1c) was worth 4. Question (2a) was worth 4 points, and (2b) 2 points (since it didn't require the full derivations). (2c) was worth 4 points, and (2d) 3 points, although these are a little convolved in their grading because they were so similar and tended to rely on a common lemma. By subtraction, then question (2d) was worth 2 points.

This homework I decided to grade positively, so you should find a number for each of the nine subparts, and the sum of those numbers should be your total grade. Obviously if I've made any math errors I'll be happy to fix them.

1 Truth tables

1.1 Large-step operational semantics

Overall, people did pretty well here. There wasn't much confusion between large-step and small-step semantics, which probably isn't too surprising since large-step is the more natural thing to do in this situation. There were basically two sorts of answers here, the "truth-tables-as-rules" solutions, which encoded the meaning of the logical operators in the creation of multiple rules for each operator, where only one rule would apply based on whether the inputs reduced to true or false, and the "mathematical-operators" solutions, which assumed the existence of "mathematical" logical operators whose behavior was already understood, and defined the truth-table semantics in terms of those operators (in much the same way we used the math $+$ and $\cdot$ operators to define our IMP semantics). Either was acceptable, although in my opinion the "truth-tables-as-rules" approach is more elegant, even though it takes up about three times as much paper. When using the "mathematical-operators" approach, it was necessary to say something to make it clear that you understood which were the truth-table operators and which were the mathematical operators, and that was worth one point of the three.

There were also a few "macro-based" solutions, which defined implication and if-and-only-if in terms of the truth-table and and or operators. This is orthogonal to the "truth-tables-as-rules" or "mathematical-operators" distinction, and was also fine.

Sample solution

This is a non-"macro-based," "truth-tables-as-rules" solution. The truth table itself is represented much like our IMP heap. Notice that it is not necessary to have four rules for all of the binary operators; and, or, and implies can be implemented with three by doing the equivalent of short-circuit evaluation. The rule for if-and-only-if uses a single rule to capture both true cases by specifying the same arbitrary truth value c for each sub-evaluation.

$$t(x) = \begin{cases} c & \text{if} \quad t = t, x = c \\ t'(x) & \text{if} \quad t = t', y = c' \\ \texttt{false} & \text{if} \quad t = \cdot \end{cases}$$

$$\frac{}{t \ ; \ c \ \Downarrow \ c} \qquad \frac{}{t \ ; \ x \ \Downarrow \ t(x)} \qquad \frac{t \ ; \ f \ \Downarrow \ \texttt{true}}{t \ ; \ \neg f \ \Downarrow \ \texttt{false}} \qquad \frac{t \ ; \ f \ \Downarrow \ \texttt{false}}{t \ ; \ \neg f \ \Downarrow \ \texttt{true}} \qquad \frac{t \ ; \ f_1 \ \Downarrow \ \texttt{true} \quad t \ ; \ f_2 \ \Downarrow \ \texttt{true}}{t \ ; \ f_1 \wedge f_2 \ \Downarrow \ \texttt{true}}$$

$$\frac{t \ ; \ f_1 \ \Downarrow \ \texttt{false}}{t \ ; \ f_1 \wedge f_2 \ \Downarrow \ \texttt{false}} \qquad \frac{t \ ; \ f_2 \ \Downarrow \ \texttt{false}}{t \ ; \ f_1 \wedge f_2 \ \Downarrow \ \texttt{false}} \qquad \frac{t \ ; \ f_1 \ \Downarrow \ \texttt{false} \quad t \ ; \ f_2 \ \Downarrow \ \texttt{false}}{t \ ; \ f_1 \vee f_2 \ \Downarrow \ \texttt{false}}$$

$$\frac{t \ ; \ f_1 \ \Downarrow \ \texttt{true}}{t \ ; \ f_1 \vee f_2 \ \Downarrow \ \texttt{true}} \qquad \frac{t \ ; \ f_2 \ \Downarrow \ \texttt{true}}{t \ ; \ f_1 \vee f_2 \ \Downarrow \ \texttt{true}} \qquad \frac{t \ ; \ f_1 \ \Downarrow \ \texttt{true} \quad t \ ; \ f_2 \ \Downarrow \ \texttt{false}}{t \ ; \ f_1 \rightarrow f_2 \ \Downarrow \ \texttt{false}}$$

$$\frac{t \ ; \ f_1 \ \Downarrow \ \texttt{false}}{t \ ; \ f_1 \rightarrow f_2 \ \Downarrow \ \texttt{true}} \qquad \frac{t \ ; \ f_2 \ \Downarrow \ \texttt{true}}{t \ ; \ f_1 \rightarrow f_2 \ \Downarrow \ \texttt{true}} \qquad \frac{t \ ; \ f_1 \ \Downarrow \ c \quad t \ ; \ f_2 \ \Downarrow \ c}{t \ ; \ f_1 \leftrightarrow f_2 \ \Downarrow \ \texttt{true}}$$

$$\frac{t \ ; \ f_1 \ \Downarrow \ \texttt{true} \quad t \ ; \ f_2 \ \Downarrow \ \texttt{false}}{t \ ; \ f_1 \leftrightarrow f_2 \ \Downarrow \ \texttt{false}} \qquad \frac{t \ ; \ f_1 \ \Downarrow \ \texttt{false} \quad t \ ; \ f_2 \ \Downarrow \ \texttt{true}}{t \ ; \ f_1 \leftrightarrow f_2 \ \Downarrow \ \texttt{false}}$$

Grading notes

- I assumed that anyone who just defined the logical operators in terms of logical operators without saying anything else about where the "right-side" operators came from or what they meant understood that they were doing a "mathematical-operators" solution, but I docked a point for failing to say something about where they came from. I accepted almost anything said about them as sufficient explanation.
- I assumed that anyone who omitted the definition of the truth-table lookup function was assuming some reasonable such function, and worked from there, but I docked a point because I think that it's important to carry these things out fully and the semantics of the truth table are important to the semantics of the whole combination. I don't think that anybody who did put down a defintion of the lookup got it wrong. I saw some solutions that didn't define a meaning for lookup in the empty heap, but I don't see that as an error when you are the one defining the semantics.
- The exception to the "undeclared assumption of mathematical logical operators" rule was the use of $=$ and $\neq$ for boolean equality (which is exactly if-and-only-if) and its negation. This tended to appear in the if-and-only-if rule, to say that the two parts had to evaluate to constants, and those constants had to be the same or not. Technically you should implement the "be the same" test by using the same variable name, and you should avoid the "not be the same" test, because the distinction between variable identity and value identity is tricky in first order logic.

1.2 Interpreter implementation

I ran everybody's interpreter on a set of 20 testcases. Two of these were somewhat complex expressions, and the others were individual tests of the different operators. Everybody's interpreters passed all the tests. I rougly allocated one point for the overall form of the interpreter (basically its type), one for functionality (which everyone got), and one for random code stuff.

Sample solution

This interpreter follows the outline of a "mathematical-operators" semantics, in that it has one major case per construct. The truth tables for the operators are encoded in the match expressions within each case. For the binary operators, we first recursively interpret the sub-expressions down to `True` or `False`, and then build a pair of the two arguments and match on that. Extensive use of the `_` to match all other cases keeps the length of the code down.

```
let rec lookup v table =
  match table with
    [] -> False
  | (x,c)::tl -> if x=v then c else lookup v tl

let rec interpret table formula =
  match formula with
    Const c -> c
  | Var v -> lookup v table
  | Not f ->
      (match interpret table f with
        True -> False
      | False -> True)
  | And(f1,f2) ->
      (match (interpret table f1, interpret table f2) with
        (True,True) -> True
      | _ -> False)
  | Or(f1,f2) ->
      (match (interpret table f1, interpret table f2) with
        (False,False) -> False
      | _ -> True)
  | Implies(f1,f2) ->
      (match (interpret table f1, interpret table f2) with
        (True,False) -> False
      | _ -> True)
  | Iff(f1,f2) ->
      (match (interpret table f1, interpret table f2) with
        (True,True) -> True
      | (False,False) -> True
      | _ -> False)
```

Grading notes

1.3 Denotational translator for truth-tables

Among the useful tools for grading this part was running `ocamlc -i` to see the types of your `denote_table` and `denote_formula` functions. You'll see this output in the log. I roughly allocated the four points for this part as one point for the overall form, two points for keeping the AST out of the denoted transformation (which admittedly can be subtle, but was important), and one point for functionality. I occasionally gave one point out of two for small AST leaks that looked more like typos in an otherwise proper outline.

Sample solution

The critical thing to see in this solution is the way we use the `let` constructs to "pull out" the recursive exploration of the AST from the function we build to denote the result. If we buried the recursive calls to `denote_formula` inside the `fun` construct, that would delay translation until runtime, making this back into an interpreter.

```
let denote_constant c =
  match c with
    True  -> true
  | False -> false

let rec denote_formula formula  =
  match formula with
    Const c ->
      let dc = denote_constant c in
      (fun t -> dc)
  | Var v ->
      (fun t -> t v)
  | Not f ->
      let df = denote_formula f in
      (fun t -> not (df t))
  | And(f1,f2) ->
      let df1 = denote_formula f1 in
      let df2 = denote_formula f2 in
      (fun t -> (df1 t) && (df2 t))
  | Or(f1,f2) ->
      let df1 = denote_formula f1 in
      let df2 = denote_formula f2 in
      (fun t -> (df1 t) || (df2 t))
  | Implies(f1,f2) ->
      let df1 = denote_formula f1 in
      let df2 = denote_formula f2 in
      (fun t -> (not (df1 t)) || (df2 t))
  | Iff(f1,f2) ->
      let df1 = denote_formula f1 in
      let df2 = denote_formula f2 in
      (fun t ->
        let a1 = (df1 t) in
        let a2 = (df2 t) in
        (a1 && a2) || ((not a1) && (not a2)))

let rec denote_table table =
  match table with
    [] -> (fun x -> false)
  | (x,c)::tl ->
      let dc  = denote_constant c in
      let dtl = denote_table tl in
      (fun y -> if x=y then dc else dtl y)
```

Grading notes

Far and away the most common error was failure to "pull up" the recursive calls as described above. This not only leaks the AST into the "denoted" version of the formula, but it also alters the order of evaluation, so that this isn't even a translator, but is really just the interpreter all over again but slightly thunked. This almost inevitably earned two points out of four by my grading guide, which is probably generous, even for something that appears to work externally.

A variation on this was to just wrap the entire interpreter in a `fun` construct (and change the name to make it sound like a denoter). This is arguably worse than the above, although logic dictated the same grading.

The most common minor error was to do the same thing (failure to pull up the recursion in this case),

but only in the `denote_table` function.

2 Nondeterministic IMP

2.1 Extension A makes IMP non-deterministic

We really wanted to see the full derivations here. Yes they are annoying and pedantic, but we want to make sure we really understand the basics before we just sweep them under the rug. Thus, the program/counter-example was worth one point, the sequence of steps was worth one, and the derivations on each step two.

Note that it really is necessary to lay out this problem in what amounts to two dimensions; there really must be multiple steps in the small-step semantics (no single step can illustrate non-determinism here) and some the steps can only be shown using at least two levels of rule application, because the non-skip rules for $s_1 \parallel s_2$ are not axioms, and you have to prove that the statements inside reduce as you need them to.

Finally, although either was sufficient, it turns out to be much easier to show programs that terminate with different results than to show programs that do or don't terminate, just because the application of the assignment rule is simpler than the looping rules. A correct solution for a loops/doesn't-loop example was probably twice as long. Also, although it was obvious and I didn't worry about it, *because* these extensions make IMP non-deterministic, meerly reaching a repeated state does not mean that the program cannot terminate, only that it might not terminate, because we can keep applying the same set of rules over and over.

Sample solution

Consider $ans := 0 \parallel ans := 1$. First derivation:

$$\dfrac{\dfrac{\dfrac{}{\cdot\ ;\ 0 \Downarrow 0}}{\cdot\ ;\ ans := 0 \ \rightarrow\ \cdot, ans \mapsto 0\ ;\ \mathsf{skip}}}{\cdot\ ;\ ans := 0 \parallel ans := 1 \ \rightarrow\ \cdot, ans \mapsto 0\ ;\ \mathsf{skip} \parallel ans := 1}$$

$$\dfrac{\dfrac{\dfrac{}{\cdot\ ;\ 1 \Downarrow 1}}{\cdot, ans \mapsto 0\ ;\ ans := 1 \ \rightarrow\ \cdot, ans \mapsto 0, ans \mapsto 1\ ;\ \mathsf{skip}}}{\cdot, ans \mapsto 0\ ;\ \mathsf{skip} \parallel ans := 1 \ \rightarrow\ \cdot, ans \mapsto 0, ans \mapsto 1\ ;\ \mathsf{skip} \parallel \mathsf{skip}}$$

$$\dfrac{}{\cdot, ans \mapsto 0, ans \mapsto 1\ ;\ \mathsf{skip} \parallel \mathsf{skip} \ \rightarrow\ \cdot, ans \mapsto 0, ans \mapsto 1\ ;\ \mathsf{skip}}$$

Second derivation:

$$\dfrac{\dfrac{\dfrac{}{\cdot\ ;\ 1 \Downarrow 1}}{\cdot\ ;\ ans := 1 \ \rightarrow\ \cdot, ans \mapsto 1\ ;\ \mathsf{skip}}}{\cdot\ ;\ ans := 0 \parallel ans := 1 \ \rightarrow\ \cdot, ans \mapsto 1\ ;\ ns := 0 \parallel \mathsf{skip}}$$

$$\dfrac{\dfrac{\dfrac{}{\cdot\ ;\ 0 \Downarrow 0}}{\cdot, ans \mapsto 1\ ;\ ans := 0 \ \rightarrow\ \cdot, ans \mapsto 1, ans \mapsto 0\ ;\ \mathsf{skip}}}{\cdot, ans \mapsto 1\ ;\ ans := 0 \parallel \mathsf{skip} \ \rightarrow\ \cdot, ans \mapsto 1, ans \mapsto 0\ ;\ \mathsf{skip} \parallel \mathsf{skip}}$$

$$\dfrac{}{\cdot, ans \mapsto 1, ans \mapsto 0\ ;\ \mathsf{skip} \parallel \mathsf{skip} \ \rightarrow\ \cdot, ans \mapsto 1, ans \mapsto 0\ ;\ \mathsf{skip}}$$

Grading notes

A common weakness, which I let slide, was omitting the bar over the top rule in the derivation. As a matter of form, a derivation tree must work back to axioms, which means that the top rules applied must all have

no requirements "over the bar." The bar is not optional.

Also, several people specified a starting point that wasn't really a stand-alone program. The typical examples were starting with a statement and a non-empty heap (for convenience, presumably, rather than putting in another assignment and showing those steps), and specifying not a single statement s but two statements s_1 and s_2 where the entire program was implicitly $s_1 \parallel s_2$. I let both of these slide with the obvious interpretations.

2.2 Extension B makes IMP non-deterministic

Sample solution

Consider $ans := 0 \parallel ans := 1$. There are many possible state sequences. Here are two that produce different answers.

$$
\begin{array}{lllll}
 & \cdot & ; & ans := 0 \parallel ans := 1 \\
\rightarrow & \cdot, ans \mapsto 0 & ; & \mathsf{skip} \parallel ans := 1 \\
\rightarrow & \cdot, ans \mapsto 0 & ; & ans := 1 \\
\rightarrow & \cdot, ans \mapsto 0, ans \mapsto 1 & ; & \mathsf{skip} \\
 & \cdot & ; & ans := 0 \parallel ans := 1 \\
\rightarrow & \cdot & ; & ans := 1 \parallel ans := 0 \\
\rightarrow & \cdot, ans \mapsto 1 & ; & \mathsf{skip} \parallel ans := 0 \\
\rightarrow & \cdot, ans \mapsto 1 & ; & ans := 0 \\
\rightarrow & \cdot, ans \mapsto 1, ans \mapsto 0 & ; & \mathsf{skip}
\end{array}
$$

Grading notes

A couple of people used the intuition from question (2e) here, rather than just using the same program from above. This works too.

2.3 Divergence under A implies divergence under B

The main challenge for this question and the next is to understand that, however we phrase it, we are doing induction in two directions. Because we are talking about the behavior of the program over many steps, we end up needing to use induction on the number of step taken. But at the same time, we can't forget that each step can involve an arbitrarily complex derivation, and we must also set up an induction on the structure of this derivation in order to show what we need for each step.

The intuition here is that extension B can always simulate any step of extension A in one or more steps. This is true because the only rule in extension A that does not directly correspond to a rule in extension B is A2, but that can be simulated by using B2, B1, and B2 again.

This problem and the next were graded collectively, since everybody ended up solving them in an essentially integrated manner. I ended up with a scheme that awarded a large fraction of the points for the concepts of the proof, plus a few for specific and general details. Basically, the concepts I looked for were

- The simulation of rule A2 using rule B2, B1, and B2 again
- The use of one or more lemmas to formalize the simulation
- Correct application of the lemmas to part (2d)
- Use of induction on n to rigorously prove that we can carry simulation to an arbitrary number of steps
- Use of structural induction to rigorously prove that a single step can really be simulated, and/or use of a prefix argument to rigorously show that the existence of a longer sequence of steps implies our definition of divergence.

All of these were worth about a point, except induction on n, which was worth two if you did it right, and one if you made a rough argument based on infinite sequences. Finally, I awarded the last point subjectively based on clarity of the exposition. My rambling rigor rants are in the grading notes section, and explain what I really wanted for some of these points.

Sample solution

This is, perhaps, the plodding and methodical version, using two lemmas to carefully separate and set up the two inductions. Note also the careful use of the definition of divergence to tie the lemmas back to the theorem.

Lemma 1 If $H\ ;\ s\ \rightarrow\ H'\ ;\ s'$ under Extension A, then $H\ ;\ s\ \rightarrow^n\ H'\ ;\ s'$ under Extension B for some $n > 0$.

Proof 1: The proof is by induction on the derivation of $H\ ;\ s\ \rightarrow\ H'\ ;\ s'$ under Extension A, proceeding by cases on the last rule used: If the last rule is ASSIGN, SEQ1, IF1, IF2, or WHILE, then the same derivation holds under Extension B, so the lemma holds with $n = 1$. This works because all of these rules are either axioms, or require only expression-related properties. None of them require anything about sub-expressions iterating to other sub-expressions. These cases remain:

SEQ2 Inverting the derivation ensures $s = s_1; s_2$ and $s' = s_1'; s_2$ for some s_1, s_2, and s_1' and there is a smaller derivation of $H\ ;\ s_1\ \rightarrow\ H'\ ;\ s_1'$ under Extension A. So by induction, $H\ ;\ s_1\ \rightarrow^n\ H'\ ;\ s_1'$ under Extension B for some $n > 0$. Therefore, by induction on n, we can show $H\ ;\ s_1; s_2\ \rightarrow^n\ H'\ ;\ s_1'; s_2$ under Extension B by using SEQ2 to derive the last step.

A1 This case is analogous to the previous one.

A2 Inverting the derivation ensures $s = s_1 \parallel s_2$ and $s' = s_1 \parallel s_2'$ for some s_1, s_2, and s_2' and there is a smaller derivation of $H\ ;\ s_2\ \rightarrow\ H'\ ;\ s_2'$ under Extension A. So by induction, $H\ ;\ s_2\ \rightarrow^n\ H'\ ;\ s_2'$ under Extension B for some $n > 0$. Therefore, by induction on n, we can show $H\ ;\ s_2 \parallel s_1\ \rightarrow^n\ H'\ ;\ s_2' \parallel s_1$ under Extension B by using B1 to derive the last step. Furthermore, we can derive $H\ ;\ s_1 \parallel s_2\ \rightarrow\ H\ ;\ s_2 \parallel s_1$ and $H'\ ;\ s_2' \parallel s_1\ \rightarrow\ H'\ ;\ s_1 \parallel s_2'$ using B2, so the lemma holds with $n + 2$.

A3 We can use B3 to take the same step, so the lemma holds with $n = 1$.

Lemma 2 If $H\ ;\ s\ \rightarrow^n\ H'\ ;\ s'$ under Extension A for some $n > 0$, then $H\ ;\ s\ \rightarrow^m\ H'\ ;\ s'$ under Extension B for some $m \geq n$

Proof 2: The proof is by induction on n, the number of steps taken under Extension A. The base case is $n = 1$, in which case Lemma 1 holds, and therefore $H\ ;\ s\ \rightarrow^m\ H'\ ;\ s'$ under Extension B for some $m > 0$. Since m is an integer (you can't take half a step) and $m > 0$, we know $m \geq 1$, so therefore $m \geq n$ and therefore lemma 2 holds for $n = 1$.

For $n > 1$, assume inductively that lemma 2 holds for $n - 1$ steps. By assumption $H\ ;\ s\ \rightarrow^n\ H'\ ;\ s'$ under Extension A, which, by definition means that

1. $H\ ;\ s\ \rightarrow\ H''\ ;\ s''$ under Extension A, and
2. $H''\ ;\ s''\ \rightarrow^{n-1}\ H'\ ;\ s'$ under Extension A.

By lemma 1, term 1 means that $H\ ;\ s\ \rightarrow^{m_1}\ H''\ ;\ s''$ under Extension B for some $m_1 > 0$, and, by the inductive hypothesis, term 2 means that $H''\ ;\ s''\ \rightarrow^{m_2}\ H'\ ;\ s'$ under Extension B for some $m_2 \geq n - 1$. These together mean that $H\ ;\ s\ \rightarrow^{m_1+m_2}\ H'\ ;\ s'$. Since $m_1 > 0$ and $m_2 \geq n - 1$, we know $m_1 + m_2 \geq n$, so lemma 2 holds for n steps.

Proof The proof of part (c) using lemma 2 is as follows. By definition, "$H; s$ diverges under Extension A" means that, for any $n > 0$, there exists an H' and s' such that $H\ ;\ s\ \rightarrow^n\ H'\ ;\ s'$ under Extension A. By lemma 2, then, there exists an $m \geq n$ such that $H\ ;\ s\ \rightarrow^m\ H'\ ;\ s'$ under Extension B. By definition (and because $m \geq n$), this means that

1. $H\ ;\ s\ \rightarrow^n\ H''\ ;\ s''$ under Extension B, and
2. $H''\ ;\ s''\ \rightarrow^{m-n}\ H'\ ;\ s'$ under Extension B.

so by term 1, we know that there exists an H'' and s'' such that $H\ ;\ s\ \rightarrow^n\ H''\ ;\ s''$ under Extension B. Since this is true for any $n > 0$, this means that $H; s$ diverges under extension B.

2.4 Termination under A implies termination under B

Sample solution

Using lemma 2 from (2c), the proof of part (d) is as follows. By definition, "$H; s$ terminates under Extension A" means that there exists an $n > 0$ such that $H\ ;\ s\ \rightarrow^n\ H'\ ;\ \mathsf{skip}$ under Extension A. By lemma 2, this means that there exists a $m > 0$ such that $H\ ;\ s\ \rightarrow^m\ H'\ ;\ \mathsf{skip}$, which means that $H; s$ terminates under Extension B.

Grading notes for (2c) and (2d)

There was a strong tendency on the part of many people to try to ignore the need for structural induction to prove what I've called lemma 1 by ignoring the rules of each derivation above the last rule, and carrying the sub-evaluations under extension A to extension B by the expedient of giving them the same name and implicitly assuming that that will work out okay. Of course it does, but the point is to prove it. To make this clear, consider the following simple and concise non-proof of lemma 1:

Non-proof 1 By assumption, $H\ ;\ s\ \rightarrow\ H'\ ;\ s'$ under extension A. The non-proof follows by cases on the rule used:

- **Any of the original IMP rules**: The same rule exists under extension B, so use it, and $H\ ;\ s\ \rightarrow\ H'\ ;\ s'$ under extension B.
- **A1**: B1 is equivalent to A1, so use it, and $H\ ;\ s\ \rightarrow\ H'\ ;\ s'$ under extension B.
- **A3**: when s_2 is skip, B3 is equivalent to A3, so use it and $H\ ;\ s\ \rightarrow\ H'\ ;\ s'$ under extension B.
- **A2**: s must have the form $s_1 \parallel s_2$, and s' must be $s_1 \parallel s_2'$. Furthermore, we know that $H\ ;\ s_2\ \rightarrow\ H'\ ;\ s_2'$ under extension A.[1]

  1. $H\ ;\ s_1 \parallel s_2\ \rightarrow\ H\ ;\ s_2 \parallel s_1$ by rule B2
  2. $H\ ;\ s_2 \parallel s_1\ \rightarrow\ H'\ ;\ s_2' \parallel s_1$ by rule B1
  3. $H'\ ;\ s_2' \parallel s_1\ \rightarrow\ H'\ ;\ s_1 \parallel s_2'$ by rule B2

  so therefore, again, $H\ ;\ s\ \rightarrow\ H'\ ;\ s'$ under extension B.

The error in this non-proof is the application of rule B1 in the last case. The requirement of rule B1 is that $H\ ;\ s_2\ \rightarrow\ H'\ ;\ s_2'$ *under extension B*, and the only way we can possibly know this is by induction on the structure of the derivation, which works because s_2 is a smaller statement than s, and therefore has a shorter derivation tree. Without this we are *absolutely* missing the case of complex nested statements.

The other common shortcoming in part (2c) was a reliance on intuition to say that if extension A can take an infinite number of steps, and extension B can simulate each of these steps, then that gives us an infinite sequence of steps in extension B. The point, however, is that that is not the technical definition of divergence. What we said in class is that $H; s$ diverges if for all $n > 0$, $H\ ;\ s\ \rightarrow^n\ H'\ ;\ s'$. Therefore we need to show the existence of a sequence of length exactly n for all $n > 0$ under extension B. Of course it's insane to think that we might be able to take say 37 steps but not be able to take only 35 steps, but proof by insanity of the contrary isn't acceptable on its own. We need to actually *say* that *because* we can take $m \geq n$ steps (and get to the same state), we can necessarily take only n steps and get *somewhere*, or else we haven't really covered all possible values of n.

2.5 Divergence under B does not imply divergence under A

This is false. The simplist counter-example is $\mathsf{skip} \parallel \mathsf{skip}$. Under Extension B, we can diverge because $H\ ;\ \mathsf{skip} \parallel \mathsf{skip}\ \rightarrow\ H\ ;\ \mathsf{skip} \parallel \mathsf{skip}$ using B2. But under Extension A, only rule A3 applies and it causes termination in one step.

[1] Of course nobody who tried this said "under extension A," or he or she would hopefully have realized the mistake.

2.6 Termination under B implies termination under A

The trick here is to define a relationship between statements to show that they are essentially equivalent under extensions A and B. The R relationship given in the hint works, and proving the two lemmas firms up the precise notion of equivalence and shows that R really does have this property. Note that you have to do it in this order; without having proved the lemmas you cannot say that R has anything to do with the heaps produced while executing the statements. The definition of R, after all, doesn't say anything about heaps.

The intuition here is that, although extension A cannot directly simulate the operation of extension B, in particular the swapping operation of rule B2, it can still perform what amounts to the same steps as extension B after a swap by using rule A2 to operate on the "other side" directly without swapping. Thus we are able to work through the same series of heaps, even though the statements may never match until the very end. In effect, rule B1 is simulated by either rule A1 or rule A2, depending on how rule B2 was used earlier in the computation. However, in contrast to parts (2c) and (2d), we cannot directly construct such a simulation, because the possiblity of nesting complicates everything (essentially the analog of the need for structural induction in those parts). Here, the "rule-swapping" is effected by the use of rules R3 and R4, which allow the equivalence property to hold regardless of the order of the equivalent subexpressions, while still requiring that the subexpressions *be* equivalent.

Sample solution

We first prove the lemmas given as hints:

Lemma 1: If $R(s, \mathsf{skip})$, then $H\ ;\ s \ \rightarrow^*\ H\ ;\ \mathsf{skip}$ under Extension A.

Proof 1: The proof is by induction on the derivation of $R(s, \mathsf{skip})$, proceeding by cases on the last rule used:

R1 Then $s = \mathsf{skip}$, so the lemma holds after 0 steps.

R2 This case is impossible.

R3 This case is impossible.

R4 This case is impossible.

R5 By inversion, $s = \mathsf{skip} \parallel s_1$ for some s_1 and $R(s_1, s_2)$. By induction, $H\ ;\ s_1 \ \rightarrow^*\ H\ ;\ \mathsf{skip}$ under Extension A. By induction on the number of steps in that sequence, we can show $H\ ;\ \mathsf{skip} \parallel s_1 \ \rightarrow^*\ H\ ;\ \mathsf{skip} \parallel \mathsf{skip}$ using A2 to derive the last step. From $H; \mathsf{skip} \parallel \mathsf{skip}$, we can use A3 to produce $H; \mathsf{skip}$.

Lemma 2: If $R(s_1, s_2)$ and $H\ ;\ s_2 \ \rightarrow\ H'\ ;\ s_2'$ under Extension B, then there exists an s_1' such that $H\ ;\ s_1 \ \rightarrow^*\ H'\ ;\ s_1'$ under Extension A and $R(s_1', s_2')$.

Proof: The proof is by induction on the derivation of $R(s_1, s_2)$, proceeding by cases on the last rule used:

R1 In this case s_1 and s_2 are the same s. We prove there exists an s_1' such that $H\ ;\ s \ \rightarrow^*\ H'\ ;\ s_1'$ under Extension A and $R(s_1', s_2')$ by an inner induction on the derivation of $H\ ;\ s \ \rightarrow\ H'\ ;\ s_2'$, proceeding by cases on the last rule used. For rules ASSIGN, SEQ1, IF1, IF2, and WHILE, we can let s_1' be s_2'. The same rule lets us derive $H\ ;\ s \ \rightarrow\ H'\ ;\ s_1'$ under Extension B and R1 lets us derive $R(s_1', s_1')$. These cases remain:

SEQ2 s has the form $s_a; s_b$ for some s_a and s_b and $H\ ;\ s_a \ \rightarrow\ H'\ ;\ s_a'$ under Extension B using a smaller derivation. So by induction there exists an n and s_a'' such that $H\ ;\ s_a \ \rightarrow^n\ H'\ ;\ s_a''$ under Extension A and $R(s_a'', s_a')$. By induction on n, we can use SEQ2 to show $H\ ;\ s_a; s_b \ \rightarrow^n\ H'\ ;\ s_a''; s_b$ under Extension A. We can use R2 and R1 to show $R(s_a''; s_b, s_a'; s_b)$. So letting s_1' be $s_a''; s_b$ suffices.

B1 (This case is analogous to the previous one.) s has the form $s_a \parallel s_b$ for some s_a and s_b and $H\ ;\ s_a \ \rightarrow\ H'\ ;\ s_a'$ under Extension B using a smaller derivation. So by induction there exists an n and s_a'' such that $H\ ;\ s_a \ \rightarrow^n\ H'\ ;\ s_a''$ under Extension A and $R(s_a'', s_a')$. By induction on n, we can use A1 to show $H\ ;\ s_a \parallel s_b \ \rightarrow^n\ H'\ ;\ s_a'' \parallel s_b$ under Extension A. We can use R3 and R1 to show $R(s_a'' \parallel s_b, s_a' \parallel s_b)$. So letting s_1' be $s_a'' \parallel s_b$ suffices.

B2 s has the form $s_a \parallel s_b$ for some s_a and s_b and s_2' is $s_b \parallel s_a$. Using R4 and R1, we can derive $R(s_a \parallel s_b, s_b \parallel s_a)$. So it suffices to take 0 steps under Extension A and let s_1' be s.

B3 (This case is analogous to the previous one.) s has the form $\mathsf{skip} \parallel s_b$ for some s_b and s_2' is s_b. Using R5 and R1, we can derive $R(\mathsf{skip} \parallel s_b, s_b$. So it suffices to take 0 steps under Extension A and let $)s_1'$ be s.

R2 In this case, s_1 and s_2 have the form $s_{1a}; s$ and $s_{2a}; s$ for some s_{1a}, s_{2a}, and s. Furthermore, $R(s_{1a}, s_{2a})$. The derivation of $H\ ;\ s_2 \ \rightarrow\ H'\ ;\ s_2'$ ends with SEQ1 or SEQ2.

In the former case, s_{2a} is skip and s_2' is s. Using Lemma 1 and $R(s_{1a}, \mathsf{skip})$, we know $H\ ;\ s_{1a} \ \rightarrow^*\ H\ ;\ \mathsf{skip}$ under Extension A, so we can use SEQ1 to show $H\ ;\ s_1 \ \rightarrow^*\ H\ ;\ s$ under Extension A. Using R1 to show $R(s, s)$ completes our obligations.

In the latter case, s_2' is $s_{2a}'; s$ for some s_{2a}' and $H\ ;\ s_{2a} \ \rightarrow\ H'\ ;\ s_{2a}'$ under Extension B. So by induction, there exists an s_{1a}' such that $H\ ;\ s_{1a} \ \rightarrow^*\ H'\ ;\ s_{1a}'$ under Extension A and $R(s_{1a}', s_{2a}')$. We can use SEQ2 to show $H\ ;\ s_1 \ \rightarrow^*\ H'\ ;\ s_{1a}'; s$ and R2 to show $R(s_{1a}'; s, s_{2a}'; s)$.

R3 In this case, s_1 and s_2 have the form $s_a \parallel s_b$ and $s_c \parallel s_d$. Furthermore, $R(s_a, s_c)$ and $R(s_b, s_d)$. The derivation of $H\ ;\ s_2 \ \rightarrow\ H'\ ;\ s_2'$ ends with B1, B2, or B3.

For B1, s_2' is $s_c' \parallel s$ for some s_c' and $H\ ;\ s_c \ \rightarrow\ H'\ ;\ s_c'$ under Extension B. So by induction, there exists an s_a' such that $H\ ;\ s_a \ \rightarrow^*\ H'\ ;\ s_a'$ under Extension A and $R(s_a', s_c')$. We can use A1 to show $H\ ;\ s_a \parallel s_b \ \rightarrow^*\ H'\ ;\ s_a' \parallel s_b$ and R3 to show $R(s_a' \parallel s_b, s_c' \parallel s_d)$.

For B2, s_2' is $s_d \parallel s_c$. Using R4, $R(s_a, s_c)$, and $R(s_b, s_d)$, we can show $R(s_1, s_2')$. So it suffices to take 0 steps under Extension A and let s_1' be s_1.

For B3, s_2' is s_d and s_c is skip. Therefore $R(s_a, \mathsf{skip})$, so Lemma 1 ensures $H\ ;\ s_a \ \rightarrow^*\ H\ ;\ \mathsf{skip}$ under Extension A, so we can use A1 to show $H\ ;\ s_a \parallel s_b \ \rightarrow^*\ H\ ;\ \mathsf{skip} \parallel s_b$. We can use R5 to show $R(\mathsf{skip} \parallel s_b, s_d)$, so letting s_1' be $\mathsf{skip} \parallel s_b$ satisfies the lemma.

R4 This case is analogous to the previous one. For case B2, we use R3 just like the previous case used R4.

R5 In this case, s_1 has the form $\mathsf{skip} \parallel s_a$ for some s_a. Furthermore, $R(s_a, s_2)$. So by induction $H\ ;\ s_a \ \rightarrow^*\ H'\ ;\ s_a'$ under Extension A for some s_a' such that $R(s_a', s_2')$. So we can use A2 to show $H\ ;\ \mathsf{skip} \parallel s_a \ \rightarrow^*\ H'\ ;\ \mathsf{skip} \parallel s_a'$ and R5 to show $R(\mathsf{skip} \parallel s_a', s_2')$. So letting s_1' be $\mathsf{skip} \parallel s_a'$ satisfies the lemma.

We now use the lemmas to prove the original claim. By Lemma 2 and induction on n, if $H\ ;\ s \ \rightarrow^*\ H'\ ;\ \mathsf{skip}$ under Extension B, then there exists an s' such that $H\ ;\ s \ \rightarrow^*\ H'\ ;\ s'$ under Extension A and $R(s', \mathsf{skip})$. (Using the lemma requires $R(s, s)$, which R1 provides.) By Lemma 1, $H'\ ;\ s' \ \rightarrow^*\ H'\ ;\ \mathsf{skip}$ under Extension A. So $H\ ;\ s \ \rightarrow^*\ H'\ ;\ \mathsf{skip}$ under Extension A.

Grading notes

There were basically three sorts of attempts at solutions here:

1. Those that used the relation R from the hint as-is (without embellishing it with any meaning outside that implied by the lemmas), and attempted to prove the lemmas more-or-less along the lines of the sample solution. Typically, the proof of the theorem using the lemmas was fairly good, and the proof of lemma 1 followed the sample solution as well. Lemma 2 was, of course, more challenging.

2. Those that applied the intuition of the relation R (that two statements fit in the relation if the two extensions can reduce them to skip while taking the respective heaps through the same series of modifications) as though it were part of the definition. The typical argument here strongly resembled a structural induction on the derivation of the proof that R(s1,s2) to show that H;s1 and H;s2 must follow the same sequence of heaps. This is essentially a stronger statement than lemma 1, so the proof of lemma 1 that followed was typically a trivial claim against the roughly-stated heap equivalence property. The proofs offered for lemma 2 tended to go strange, probably because under this intuition lemma 2 isn't really necessary.

This formulation also had the interesting property of confusing what was happening under extension A and what was happening under extension B. Some people ended up more-or-less proving that R was heap equivalence for statements executing in under same extension, which isn't exactly useful.

3. Those that tried to wing it based on the overall intuition of simulating rule $B2$ by not doing anything but flipping a bit somewhere to control whether rule $A1$ or $A2$ is used to simulate rule $B1$. The weakness in this approach is in accounting for the fact that the program is more than just a single non-determinism operator applied to purely-deterministic subparts, and that the derivations that allow each step of the small-step semantics may be arbitrarily complex.

Some people tried to work both terminating and diverging programs into their proofs, which should be a danger sign right off the bat because we just got through proving that divegence under extension B does not imply divergence under extension A.