The University of Melbourne
School of Computing and Information Systems
COMP30026 Models of Computation

Assignment 1 sample solutions and comments

Challenge 1

a. To show that $F \vee G \vee G' \vee H$ is not valid, it suffices to find a counter-model, that is, an interpretation that makes the formula false. Take as domain the set $\{0, 1\}$ and interpret $P(x, y)$ as "at least one of x and y is 0." That is, if we want to list the pairs (x, y) that are in the relation P, they are $(0, 0)$, $(0, 1)$, and $(1, 0)$. This interpretation makes each disjunct false; namely

- F is false, since $P(0, 0)$ holds.
- G is false, since $P(1, 0)$ and $P(0, 1)$ hold, but $P(1, 1)$ does not.
- G' is false, since $P(0, 1)$ and $P(1, 0)$ hold, yet $\neg P(0, 0)$ is false.
- H is false, since $P(0, 0)$ is false.

b. To show that $F \wedge G' \wedge H$ is satisfiable, we need to provide a model. A simple interpretation in this case is to consider the domain $\{p, s, r\}$ with the reading "paper", "scissors", and "rock", and then interpret $P(x, y)$ as "x prevails over y according to the rules of the paper-scissors-rock game." According to the rules, $P(p, r)$, $P(s, p)$, and $P(r, s)$ are true, and for all other combinations (x, y), $P(x, y)$ is false. It is easy to check that this interpretation satisfies each of F, G' and H.

c. To show that $(F \wedge G) \Rightarrow H$ is valid, we show that its negation (that is, $\neg((F \wedge G) \Rightarrow H) \equiv F \wedge G \wedge \neg H$) is unsatisfiable. We do this by deriving $\bot$ from it, using resolution. First we convert $F \wedge G \wedge \neg H$ to clausal form. F contributes a single clause:

$$\neg P(v, v)$$

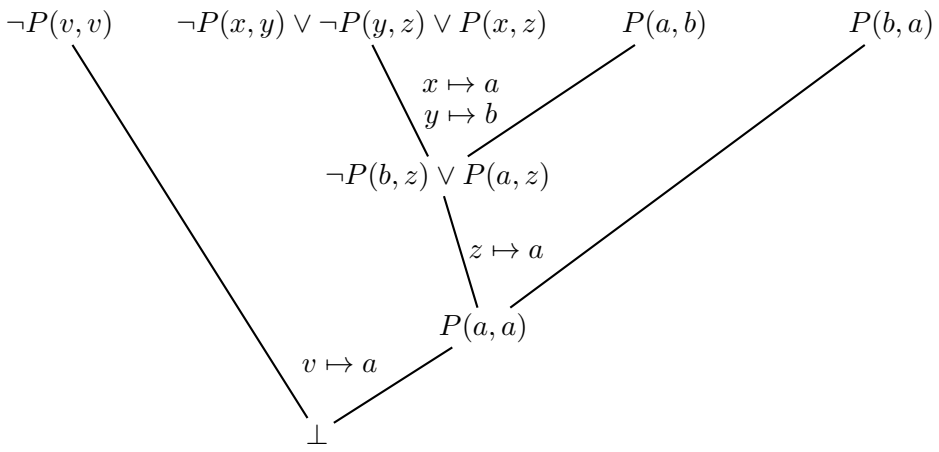
G also contributes a single clause:

$$\neg P(x, y) \vee \neg P(y, z) \vee P(x, z)$$

Finally, $\neg H$ is $\exists x \exists y (P(x, y) \wedge P(y, x))$. After Skolemization, this contributes two clauses:

$$P(a, b)$$
$$P(b, a)$$

Now a resolution refutation can proceed as follows:



Challenge 1 feedback, by Harald

For 1a we need a counter-model, or, alternatively, we can proceed as many do and negate the formula, and then find a model for the negated formula. Those are equally good approaches. For 1b we need a model.

a. A (counter-)model is an interpretation, and an interpretation has no business talking about variables. In the given case we just need a domain D and a subset of $D \times D$, to give meaning to the binary predicate P.

b. Some people trip over the quantifiers. Note that:

   (a) Although all variables are universally quantified, we cannot just remove the quantifiers; that will create problems if we need to negate (parts of) formulas later.

   (b) The universal quantifier does not distribute over $\vee$, so we cannot just move all quantifiers to the front. The quantified variables are all different, in spite of the fact that all of F, G, G' and H talk about x, for example.

   (c) Being familiar with the correct "rules of passage" for the quantifiers is very useful.

c. Again, it is important to be very careful with the quantifiers. I saw "solutions" that removed quantifiers before negating the consequent, and thus missed some Skolem constants.

   One approach, which happens to work for this challenge, is to start from the clauses $\neg P(v, v)$ (from F) and $\neg P(x, y) \vee \neg P(y, z) \vee P(x, z)$ (from G) and then derive the clausal form for H by resolution. The resolvent here comes out as $\neg P(v, y) \vee \neg P(y, v)$, which is just what we wanted.

   However, *we strongly advise against using this strategy*, because resolution does not promise to generate all logical consequences of a given set of clauses. For a very simple example, $P \vee Q$ is a logical consequence of the two clauses P and Q, but it won't be produced by resolution. Moreover, with clausal form we don't have existential quantifiers available (indeed that is why we had to come up with the trick of Skolemization in the *translation* process). A resolution step cannot introduce new constants and functions like that. Hence, if you start with a single clause like $P(x)$, you don't have a way of generating a resolvent that expresses $\exists x(P(x))$. And yet, $\forall x(P(x)) \models \exists x(P(x))$. Whatever Ψ we want to establish as a logical consequence of some Φ, if Ψ has existential quantifiers (like $\forall u \exists v \forall x \exists y(P(u, v, x, y))$), we can't just use resolution to produce a resolvent that captures that Ψ. But we can always negate Ψ, translate to clausal form, and then use resolution to establish $\Phi \models \Psi$. So that is the superior route.

Challenge 2

a. These are the four statements, in predicate logic form:

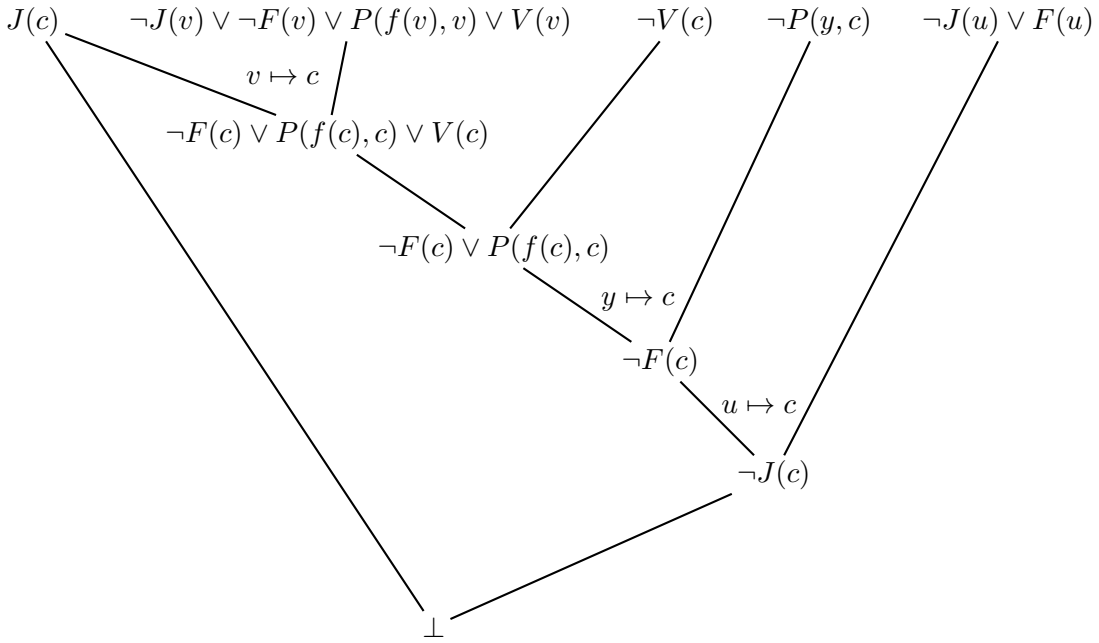
$$
\begin{aligned}
S_1: &\quad \forall x\ (J(x) \Rightarrow F(x)) \\
S_2: &\quad \exists y\ (J(y) \wedge E(a,y)) \\
S_3: &\quad \forall x\ ((J(x) \wedge F(x) \wedge \forall\ y(P(y,x) \Rightarrow E(y,x))) \Rightarrow V(x)) \\
S_4: &\quad \forall x\ ((J(x) \wedge \neg\exists y\ (P(y,x))) \Rightarrow V(x))
\end{aligned}
$$

b. And here are the first three after translation to clausal form:

$$
\begin{aligned}
S_1: &\quad \text{One clause:} && \neg J(u) \vee F(u) \\
S_2: &\quad \text{Two clauses:} && J(b) \\
&\quad \text{and} && E(a,b) \\
S_3: &\quad \text{Two clauses:} && \neg J(v) \vee \neg F(v) \vee P(f(v),v) \vee V(v) \\
&\quad \text{and} && \neg J(x) \vee \neg F(x) \vee \neg E(f(x),x) \vee V(x)
\end{aligned}
$$

c. The negation of S_4 is $\exists x\ (J(x) \wedge \forall y\ (\neg P(y,x)) \wedge \neg V(x))$. In clausal form this gives three clauses: $J(c)$, $\neg P(y,c)$, and $\neg V(c)$.

d. Altogether we have eight clauses, but the proof that S_4 follows from the rest only requires five of the clauses, namely the three from $\neg S_4$ and two of the five that we generated from S_1–S_3:



Challenge 2 feedback, by Dongge

The following were common mistakes that people made:

a. Allowing ambiguity by leaving out parentheses. Instead of:

$$\forall x \left(P(x) \Rightarrow Q(x) \right)$$

people wrote:

$$\forall x P(x) \Rightarrow Q(x)$$

b. Confusing $\Rightarrow$ with $\wedge$. Some students used the wrong one when translating English statement to formula.

c. Ignoring the presence of implication/negation while removing quantifiers. For example:

$$\forall x \left(\left(\forall y P(y) \right) \Rightarrow Q(x) \right)$$

$$\forall x \left(\left(P(y) \right) \Rightarrow Q(x) \right)$$

OR

$$\neg \forall x P(x)$$

$$\neg P(x)$$

d. Negating incorrectly. For example, while negating S:

$$S = \forall x H(x)$$

Instead of:

$$\neg S = \neg \left(\forall x H(x) \right)$$

many students wrote:

$$\neg S = \forall x \neg H(x)$$

e. Negating too late. Some negated formula **S4** *after* transforming it to clausal form, which misses flipping $\forall$ to $\exists$.

f. Getting the concept of *clause* wrong. For example, clausifying $P(x) \wedge Q(x)$ as $\{\{P(x), Q(x)\}\}$.

g. Using illegal mappings. For example, $f(x) \mapsto a$, $b \mapsto a$.

h. Failing to replace all occurrences of a variable when applying a substitution. For example:

$$\frac{\{P(x), Q(x)\} \quad \{\neg P(a)\}}{\{Q(x)\}}$$

i. Jumping prematurely to a conclusion when refutation did not reach contradiction. For example, instead of using resolution tree to show the negation of the formula in **S4** is unsatisfiable given **S1-S3**, some people attempt to prove **S4** is a logical consequence of **S1-S3** by showing they can get formula **S4** by simplifying the conjunction of formulas **S1-S3** in the tree.

Challenge 3

a. We have:

$$\begin{aligned} \boldsymbol{f_a}(0,0) &= (1,0) \\ \boldsymbol{f_a}(0,1) &= (1,1) \\ \boldsymbol{f_a}(1,0) &= (0,1) \\ \boldsymbol{f_a}(1,1) &= (0,0) \end{aligned}$$

Since every $\boldsymbol{y} \in \mathcal{B}^2$ appears in the output set, and we can verify that no distinct $\boldsymbol{x}_1, \boldsymbol{x}_2 \in \mathcal{B}^2$ map to the same $\boldsymbol{y} \in \mathcal{B}^2$, we can conclude that every vector $\boldsymbol{y} \in \mathcal{B}^n$ has a unique input vector $\boldsymbol{x}$ such that $\boldsymbol{f_a}(\boldsymbol{x}) = \boldsymbol{y}$. Hence $\boldsymbol{f_a}$ is reversible.

The inverse function to $\boldsymbol{f_a}$ is $\boldsymbol{g_a}$ defined by $\boldsymbol{g_a}(b_1, b_2) = (\neg b_1, b_1 \Leftrightarrow b_2)$. To witness,

$$\boldsymbol{g_a}(\boldsymbol{f_a}(b_1, b_2)) = \boldsymbol{g_a}(\neg b_1, b_1 \oplus b_2) = (\neg\neg b_1, \neg b_1 \Leftrightarrow (b_1 \oplus b_2) = (b_1, b_2)$$

b. We have $\boldsymbol{f_b}(0,0,0) = \boldsymbol{f_b}(1,1,1) = (0,0,0)$, so $\boldsymbol{f_b}$ cannot be reversible. Alternatively, there is no $\boldsymbol{x} \in \mathcal{B}^3$ such that $\boldsymbol{f_b}(\boldsymbol{x}) = (0,0,1)$.

c. The truth table for $\boldsymbol{f_c}$ is this:

b_1	b_2	b_3	$\boldsymbol{f_c}(b_1, b_2, b_3)$		
0	0	0	0	1	0
0	0	1	1	0	0
0	1	0	0	1	1
0	1	1	1	0	1
1	0	0	1	1	1
1	0	1	0	0	1
1	1	0	1	1	0
1	1	1	0	0	0

We note that the eight bit-vectors on the right are all different. Another way of saying this is to say that the rows in the right-hand part of the table is a permutation of the rows on the left, that is, a permutation of the eight possible truth assignments. Furthermore, every vector in $\mathcal{B}^3$ appears in the output set, hence $\boldsymbol{f_c}$ is reversible. This is its inverse:

$$\boldsymbol{g_c}(b_1, b_2, b_3) = (b_1 \Leftrightarrow b_2, b_1 \Leftrightarrow (b_2 \oplus b_3), \neg b_2)$$

d. Let τ_n be the list $\langle (0, \ldots, 0, 0), (0, \ldots, 0, 1), (0, \ldots, 1, 0), \ldots, (1, \ldots, 1, 1) \rangle$ of n-tuples of Boolean values. Note that this list of n-tuples has length 2^n.

An n-place Boolean vector function $f : \mathcal{B}^n \to \mathcal{B}^n$ is reversible iff $\langle f(v) \mid v \in \tau_n \rangle$ is a permutation of τ_n (of the same length). Since the length of τ_n is the size of $\mathcal{B}^n$, there are $|\mathcal{B}^n|! = 2^n!$ different permutations.

Now consider all Boolean vector functions $f : \mathcal{B}^n \to \mathcal{B}^n$. We count the number of choices involved in determining a unique such function.

If a function has k inputs and h possible outputs to choose from, we must assign one of h possible outputs for each of the k inputs. Hence there are h^k choices made in determining a function. Since a change in any of these choices produces a distinct function, there are h^k such functions.

Since all n-place Boolean vector functions have 2^n inputs and 2^n possible outputs, there are $(2^n)^{2^n} = 2^{n2^n}$ distinct functions from $\mathcal{B}^n$ to $\mathcal{B}^n$. So the fraction of reversible functions is $\frac{2^n!}{2^{n2^n}}$.

Challenge 3 feedback, by Billy

a, c. Almost everyone recognised that we needed to verify that the outputs of the function were distinct from each other—otherwise we would have an output without a unique input—however almost no one recognised that the statement of reversibility is a little stronger than this. This is because it's about the full output set $\mathcal{B}^n$—not just the set of outputs of the function. So in addition to verifying that the function outputs are unique, we should verify that they are actually *all there*. It turns out that this is always true when the input and output set are the same (and finite)—but this is a subtlety that you should keep in mind in your proofs.

Another common method of proof here was manually deconstructing the outputs of a function to extract the original inputs as a step by step procedure. This method is an excellent way of figuring out how to "reverse" the function, but is a little messy to present as a proof. Instead, you should collect all of the algebraic manipulation you've done into a single function from $\mathcal{B}^n$ to $\mathcal{B}^n$, and show that it reverses outputs back into inputs (as we did in the solution of part (a)).

Related to this, when you are trying to show that we can determine the value of a boolean variable a, given that we know $(a \oplus b)$ and b, instead of discussing every possible case of the different values they could have, just compute it like this:

$$(a \oplus b) \oplus b \equiv a \oplus b \oplus b \equiv a \oplus \boldsymbol{t} \equiv a.$$

b. Many people identified that **all** of the function's outputs had more than one input mapping to it and explicitly indicated which inputs map to the same output. I would advise just providing *one* of these counter-examples (providing more doesn't make your proof any more correct). Although there is value in understanding the problem space better, you introduce more potential for error and possibly spend longer than necessary on a question in an exam context.

Another concern is that you might have been misled into thinking that a function is only reversible if *every* output has more than one possible corresponding input. This is not true. Even if we can "reverse" some inputs, the existence of a single output we can't reverse is enough to refute reversibility (from the definition).

d. I think many people found this question difficult to approach, due to a misleading intuition about what a function *is*. This intuition is that a function is some kind of algebraic expression—like $f(x) = 2x + 4$—that tells you how to nicely turn an input into an output. This kind of presentation of a function is convenient, but it is not a fundamental part of the data of a function. Not all functions can be expressed like this!

Recall that a function $f : X \to Y$ is precisely a relation on X, Y, i.e., a subset of $X \times Y$, such that every $\boldsymbol{x} \in X$ is related to exactly one $\boldsymbol{y} \in Y$ (related meaning the tuple $(\boldsymbol{x}, \boldsymbol{y})$ appears in the set). It is purely notation to write $f(4) = 12$—this just means $(4, 12) \in f$.

This means that if you take **any** subset $F \subseteq X \times Y$ with each $\boldsymbol{x} \in X$ appearing in exactly one of the tuples of F—you have a function! This is why we focus on the assignments of outputs to inputs—rather than different algebraic expressions—when we are trying to count all of the different possible functions. We want to avoid missing any functions that are too difficult to express algebraically, and also avoid double counting functions that look different, but are actually the same function (like $f(x) = 3x$ and $g(x) = 2x + x$).

Challenge 4

a. The formula is equivalent to $\neg Q$.

b. The formula is equivalent to $P \Rightarrow (Q \Rightarrow R)$ (also $Q \Rightarrow (P \Rightarrow R)$) using only $\Rightarrow$.

c. The formula is equivalent to $P \Leftrightarrow Q$ (also $Q \Leftrightarrow P$) using only $\Leftrightarrow$.

Challenge 4 feedback, by Matt

Note that the parser introduced in worksheet 1 requires expressions to be heavily parenthesised. For example, many students submitted `a = parse "~Q"` which actually causes a runtime error, because the parser expects negated expressions to be parenthesised. This was missed by the visible tests, so we didn't deduct any marks. A working expression would be `a = parse "(~Q)"`. It's important to test your solutions carefully before submitting on Grok, because in general the visible tests we provide are not comprehensive. Keep this in mind while completing assignment 2.

Challenge 5

A sample solution with detailed commentary is available on Grok. Please take any questions about the solution's method or Haskell code to the LMS Discussion Board.

Challenge 5 feedback, by Matt

We calculated marks for this challenge based on the number of test cases passed out of the 10 test cases visible before the deadline (including those 'hidden' tests, which are 'visible' in that you can see the result) and 10 additional, invisible test cases. We have now revealed your results on all 20 test cases, so you might like to take another look at your submission results on Grok.

The overall class performance on this challenge proves that it was a tough one. This challenge was designed to push your propositional modelling and Haskell programming skills. Well done to those who passed most or all of the tests for this challenge.

One point for improvement for many students who passed most of the test cases is that the efficiency of the generated formula is quite important. A SAT (propositional satisfiability) solver runs in time exponential in the number of variables in the worst case. Since we use the Tseytin transform as a preprocessing step, and this introduces new variables for each connective in the original formula, we must also worry about the size of the original formula. I would say the key to creating an efficient formula in this case is to take advantage of the expressive power of the exclusive or connective ($\oplus$) for modelling the pressing dynamics of the puzzle (see sample solution for details). Our final two test cases tested puzzle boards with `n=5` or `n=6`. The suggested solution on Grok works even for some puzzles with `n=7`.

Challenge 6

The top 25% of solutions used 35 or fewer gates. The top 50% of solutions used 46 or fewer gates. The shortest circuit used just 23 gates. Well done to everyone who completed this challenge.