Mini-Quiz 6

1. Write your name:

2. (Rosen, Sec. 2.6, Ex. 29) Let

$$A = \begin{bmatrix} 1 & 0 & 1 \\ 1 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$B = \begin{bmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 0 & 1 \end{bmatrix}$$

Find the following:

(a) $A \vee B$
(b) $A \wedge B$
(c) $A \odot B$

Solution:

(a) $A \vee B$:

$$\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 0 & 1 \end{bmatrix}$$

(b) $A \wedge B$:

$$\begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

(c) $A \odot B$:
Note that $\odot$ means boolean product. Use matrix multiplication, but cap the values at one: $1 + 1 = 1 \cdot 1 = 1$

$$\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 0 & 1 \end{bmatrix}$$

Tutorial 6 Problems

Problem 1 Is a Harvard degree really worth more than an MIT degree?! Let us say that a person with a Harvard degree starts with \$40,000 and gets a \$20,000 raise every year after graduation, whereas a person with an MIT degree starts with \$30,000, but gets a 20% raise every year. Assume inflation is a fixed 8% every year. That is, \$1.08 a year from now is worth \$1.00 today. (You'll need a calculator to get final answers; if one is not available, it's ok to express the answer as a closed form numerical expression.)

(a) How much will a person with a Harvard degree be making in the n^{th} year?

Solution:

$$40000 + 20000(n-1)$$

Note that in the n^{th} year, you have not yet gotten the n^{th} raise. You should always check your boundary cases – in this case, the salary in the first year.

(b) How much will a person with an MIT degree be making in the n^{th} year?

Solution:

$$30000(1.2^{n-1})$$

(c) How much is a Harvard degree worth today if the holder will work for n years following graduation? *Hint*: In Lecture 8 a closed formula for $\sum_{i=0}^{n} ir^i$ was derived.

Solution:

One dollar after year i is worth r^i in today's currency, where

$$r = \frac{1}{1.08} = 0.925\,925\,925\ldots.$$

So

$$\begin{aligned}\text{Hvd}_n &= \sum_{i=0}^{n-1}(40000+20000i)r^i \\ &= 40000\sum_{i=0}^{n-1}r^i + 20000\sum_{i=0}^{n-1}ir^i.\end{aligned}$$

But we know from Lecture Notes 8 that

$$\sum_{i=0}^{n} ir^i = \frac{r-(n+1)r^{n+1}+nr^{n+2}}{(1-r)^2},$$

so

$$\text{Hvd}_n = 40000\frac{(1-r^n)}{1-r} + 20000\frac{(r-nr^n+(n-1)r^{n+1})}{(1-r)^2},$$

and for $n = 20$,

$$\text{Hvd}_{20} = \$1,916,483.$$

(As indicated in the first two parts, we have avoided counting the $n+1$ year, in which we do not actually work. We also could have started with:

$$\begin{aligned}\text{Hvd}_n &= \sum_{i=1}^{n}(\textit{salary-in-ith-year})\cdot(\textit{inflation-adj-for-ith-year}) \\ &= \sum_{i=1}^{n}(40000+20000(i-1))r^{i-1} \\ &= \sum_{i=0}^{n-1}(40000+20000i)r^i\end{aligned}$$

which is equivalent. A good thing to check for here, besides the boundary cases, is the number of items you are summing. If the summation were from 0 to n, that would be $n+1$ years – clearly a mistake.)

(d) How much is an MIT degree worth in this case?

Solution:

$$\begin{aligned}\text{MIT}_n &= 30000\sum_{i=0}^{n-1}1.2^i r^i \\ &= 30000\sum_{i=0}^{n-1}(1.2r)^i \\ &= \frac{30000(1-(1.2r)^n)}{1-1.2r}\end{aligned}$$

and for $n = 20$,

$$\text{MIT}_{20} \quad = \quad \frac{30000(1-(1.2r)^{20})}{1-1.2r} = \$1,950,821.$$

(e) If you plan to retire after twenty years, which degree would be worth more?

Solution:

The MIT degree is more valuable – as if you couldn't guess `:-)` – but only by about 2%. But if you were persistent enough to work 30 years, you would find the MIT degree was worth more than twice as much.

Problem 2 There are "sum" and "difference" operators that play the same role in discrete math as integral and derivative operators place in calculus. In particular, the difference operator Δ is defined by

$$\Delta f(x) = f(x+1) - f(x)$$

and the summation operator

$$\sum_a^b g(x)\delta x = \sum_{x=a}^{b-1} g(x)$$

This sum operator is a bit annoying, as many sloppy mathematicians will use the left hand side to denote a sum up to b, rather than $b-1$. But let's work with it for today.

(a) Suppose that $g(x) = \Delta f(x)$. Prove that

$$\sum_a^b g(x)\delta x = f(b) - f(a)$$

Solution:
Loosely,

$$\begin{aligned}
\sum_a^b g(x)\delta x &= \sum_{x=a}^{b-1} g(x) \\
&= \sum_{x=a}^{b-1} [f(x+1) - f(x)] \\
&= \sum_{x=a}^{b-1} f(x+1) - \sum_{x=a}^{b-1} f(x) \\
&= \sum_{x=a+1}^{b} f(x) - \sum_{x=a}^{b-1} f(x) \\
&= f(b) + \sum_{x=a+1}^{b-1} f(x) - \sum_{x=a+1}^{b-1} f(x) - f(a) \\
&= f(b) - f(a)
\end{aligned}$$

A more formal proof can be done using induction. Note, however, that using induction does not help if you cannot guess the correct formula. Manipulating the summations will give you a feel for that.

Fix arbitrary a, and induct over the values of b. Take as the base case $b = a$, for which the given equation trivially holds – there are no terms in the summation, and $f(a) - f(a) = 0$.

Now, assuming $\sum_a^b g(x)\delta x = f(b) - f(a)$, prove $\sum_a^{b+1} g(x)\delta x = f(b+1) - f(a)$.

$$\begin{aligned}
\sum_a^{b+1} g(x)\delta x &= \sum_a^{b} g(x) \\
&= g(b) + \sum_a^{b-1} g(x) \\
&= g(b) + \sum_a^{b} g(x)\delta x \\
&= g(b) + f(b) - f(a) \quad (\textbf{IH}) \\
&= \Delta f(b) + f(b) - f(a) \\
&= f(b+1) - f(b) + f(b) - f(a) \\
&= f(b+1) - f(a)
\end{aligned}$$

(b) Suppose that $h(n) = \sum_{k=0}^{n} f(k)$. What is $\Delta h(n)$?

Solution:

$$\begin{aligned}
\Delta h(n) &= h(n+1) - h(n) \\
&= \sum_{k=0}^{n+1} f(k) - \sum_{k=0}^{n} f(k) \\
&= f(n+1) + \sum_{k=0}^{n} f(k) - \sum_{k=0}^{n} f(k) \\
&= f(n+1)
\end{aligned}$$

(c) Prove that

$$\Delta(u(x)v(x)) = u(x)\Delta v(x) + v(x+1)\Delta u(x)$$

Solution:
We can work left to right and make it look like magic, introducing a zero-sum pair out of the void. Or we can work right to left:

$$\begin{aligned}
u(x)\Delta v(x) + v(x+1)\Delta u(x) &= u(x)\left(v(x+1) - v(x)\right) + v(x+1)\left(u(x+1) - u(x)\right) \\
&= u(x)v(x+1) - u(x)v(x) + v(x+1)u(x+1) - v(x+1)u(x) \\
&= v(x+1)u(x+1) - u(x)v(x) \\
&= \delta(u(x)v(x))
\end{aligned}$$

(d) The *falling power* function is defined by the equation

$$x^{\underline{m}} = x(x-1)\cdots(x-m+2)(x-m+1).$$

In other words, just like x^m, it is a product of m terms, but they fall away from x. Prove

$$\Delta x^{\underline{m}} = m x^{\underline{m-1}}$$

Solution:

$$\begin{aligned}
\Delta x^{\underline{m}} &= (x+1)^{\underline{m}} - x^{\underline{m}} \\
&= (x+1)(\mathbf{x})(\mathbf{x-1})\cdots(\mathbf{x-m+2}) \\
&\quad - \quad (\mathbf{x})(\mathbf{x-1})\cdots(\mathbf{x-m+2})(x-m+1) \\
&= (x+1)x^{\underline{m-1}} - (x-m+1)x^{\underline{m-1}} \\
&= m x^{\underline{m-1}}
\end{aligned}$$

(e) Find the value of

$$\sum_{x=a}^{b} x^{\underline{m}}$$

Solution:
We can use

$$\delta x^{\underline{m}} = m x^{\underline{m-1}} \rightarrow x^{\underline{m}} = \frac{1}{m+1}\Delta x^{\underline{m+1}}$$

$$
\begin{aligned}
\sum_{x=a}^{b} x^{\underline{m}} &= \sum_{x=a}^{b} \frac{1}{m+1} \Delta x^{\underline{m+1}} \\
&= \frac{1}{m+1} \sum_{x=a}^{b} \Delta x^{\underline{m+1}} \\
&= \frac{1}{m+1} \sum_{x=a}^{b} \left[(x+1)^{\underline{m+1}} - x^{\underline{m+1}} \right] \\
&= \frac{1}{m+1} \left[\sum_{x=a}^{b} (x+1)^{\underline{m+1}} - \sum_{x=a}^{b} x^{\underline{m+1}} \right] \\
&= \frac{1}{m+1} \left[\sum_{x=a+1}^{b+1} x^{\underline{m+1}} - \sum_{x=a}^{b} x^{\underline{m+1}} \right] \\
&= \frac{1}{m+1} \left[(b+1)^{\underline{m+1}} + \sum_{x=a+1}^{b} x^{\underline{m+1}} - a^{\underline{m+1}} - \sum_{x=a+1}^{b} x^{\underline{m+1}} \right] \\
&= \frac{(b+1)^{\underline{m+1}} - a^{\underline{m+1}}}{m+1}
\end{aligned}
$$

Problem 3 Consider the Strong Caching system given in lecture as a labeled state machine:

Caching system (strong)

The system is composed of a main memory store that is accessed by two 'clients'. Each client accesses the memory through a cache. When the memory is updated, the caches are cleaned out.

State machine:

- Q: For every $x \in addresses$:
  - $memory(x)$, an element of $data$
  - $cache_1(x)$, an element of $data \cup \{null\}$
  - $cache_2(x)$, an element of $data \cup \{null\}$

  And a number of doubled structures (one for each client/cache):
  - req_1, an element of $\{\text{“}read\text{”}, \text{“}write\text{”}\} \cup \{null\}$
  - $reqaddr_1$, an element of $addresses \cup \{null\}$ (used for both reads and writes)
  - $reqval_1$, an element of $data \cup \{null\}$ (used for writes only)
  - ret_1, an element of $\{\text{“}OK\text{”}, null\} \cup data$
  - req_2, an element of $\{\text{“}read\text{”}, \text{“}write\text{”}\} \cup \{null\}$
  - $reqaddr_2$, an element of $addresses \cup \{null\}$ (used for both reads and writes)
  - $reqval_2$, an element of $data \cup \{null\}$ (used for writes only)
  - ret_2, an element of $\{\text{“}OK\text{”}, null\} \cup data$
- Q_0: *memory* arbitrary, caches all *null*, all other components *null*.
- L: All are for $i \in \{1, 2\}$ (each client). Note that, for each label, the italicized text is just a description of the intended purpose.

  Input labels:

  req-read$_\mathbf{i}$ (x), $x \in addresses$
  an input from client i requesting to read the value in address x from $cache_i$.

  req-write$_\mathbf{i}$ (x, v), $x \in addresses$, $v \in data$
  a request by client i to write value v in location x in memory (not cache).

  Internal labels:

  comp-read$_\mathbf{i}$ (x), $x \in addresses$
  actually performed the read that was requested and stored the result.

  comp-write$_\mathbf{i}$ (x, v), $x \in addresses$, $v \in data$
  actually performed the write that was requested.

  copy$_\mathbf{i}$ $(\mathbf{x})$, $x \in addresses$
  copied the value at x from memory into $cache_i$.

  drop$_\mathbf{i}$ $(\mathbf{x})$, $x \in addresses$
  dropped the value at x from $cache_i$.

Output labels:

ret-read$_i$ (v), $v \in data$
signal result of a read request and computation for the value at x.
ret-write$_i$ $(\text{“}OK\text{”})$
signal completion of a write request and computation.

- δ:

req-read$_i$ (x)
Can occur anytime
if $req_i = null$ then
  $req_i := \text{“}read\text{”}$
  $reqaddr_i := x$

req-write$_i$ (x, v)
Can occur anytime
if $req_i = null$ then
  $req_i := \text{“}write\text{”}$
  $reqaddr_i := x$
  $reqval_i := v$

ret-read$_i$ (v)
Can occur if $retval_i \in values$
  $req_i := null$
  $reqaddr_i := null$
  $reqval_i := null$
  $retval_i := null$

ret-write$_i$ $(\text{“}OK\text{”})$
Can occur if $retval_i = \text{“}OK\text{”}$
  $req_i := null$
  $reqaddr_i := null$
  $reqval_i := null$
  $retval_i := null$

comp-read$_i$ (x)
Can occur if $req_i = \text{“}read\text{”}$, $reqaddr_i = x$, $cache_i(x) \neq null$, and $retval_i = null$
  $retval_i := cache_i(x)$

comp-write$_i$ (x, v)
Can occur if $req_i = \text{“}write\text{”}$, $reqaddr_i = x$, $reqval_i(x) = v$, and $retval_i = null$
  $memory(x) := v$
  $retval_i := \text{“}OK\text{”}$
  $cache_1(x) := null$
  $cache_2(x) := null$

copy$_i$ (x)
Can occur anytime
  $cache_i(x) := memory(x)$

drop$_i$ (x)
Can occur anytime
  $cache_i(x) := null$

Also consider a simple Centralized Memory system as a labeled state machine:

Centralized memory system

The system is composed of a main memory store that is accessed by two 'clients'. Each client accesses the memory directly.

State machine:

- Q: For every $x \in addresses$:
  - $memory(x)$, an element of $data$

  And a number of doubled structures (one for each client):
  - req_1, an element of $\{\text{“}read\text{”}, \text{“}write\text{”}\} \cup \{null\}$
  - $reqaddr_1$, an element of $addresses \cup \{null\}$ (used for both reads and writes)
  - $reqval_1$, an element of $data \cup \{null\}$ (used for writes only)
  - ret_1, an element of $\{\text{“}OK\text{”}, null\} \cup data$
  - req_2, an element of $\{\text{“}read\text{”}, \text{“}write\text{”}\} \cup \{null\}$
  - $reqaddr_2$, an element of $addresses \cup \{null\}$ (used for both reads and writes)
  - $reqval_2$, an element of $data \cup \{null\}$ (used for writes only)
  - ret_2, an element of $\{\text{“}OK\text{”}, null\} \cup data$
- Q_0: *memory* arbitrary, all other components *null*.
- L: All are for $i \in \{1, 2\}$ (each client). Note that, for each label, the italicized text is just a description of the intended purpose.

  Input labels:

  req-read$_i$ (x), $x \in addresses$
  an input from client i requesting to read the value in address x from memory.

  req-write$_i$ (x, v), $x \in addresses$, $v \in data$
  a request by client i to write value v in location x in memory.

  Internal labels:

  comp-read$_i$ (x), $x \in addresses$
  actually performed the read that was requested and stored the result.

  comp-write$_i$ (x, v), $x \in addresses$, $v \in data$
  actually performed the write that was requested.

  Output labels:

  ret-read$_i$ (v), $v \in data$
  signal result of a read request and computation for the value at x.

  ret-write$_i$(“OK”)
  signal completion of a write request and computation.

- δ:

  req-read$_i$ (x)
  Can occur anytime
  if $req_i = null$ then
  $req_i :=$ "$read$"
  $reqaddr_i := x$

  req-write$_i$ (x, v)
  Can occur anytime
  if $req_i = null$ then
  $req_i :=$ "$write$"
  $reqaddr_i := x$
  $reqval_i := v$

  ret-read$_i$ (v)
  Can occur if $retval_i \in values$
  $req_i :=null$
  $reqaddr_i := null$
  $reqval_i :=null$
  $retval_i := null$

  ret-write$_i$("OK")
  Can occur if $retval_i =$ "OK"
  $req_i :=null$
  $reqaddr_i := null$
  $reqval_i :=null$
  $retval_i := null$

  comp-read$_i$ (x)
  Can occur if $req_i =$ "$read$", $reqaddr_i = x$, and $retval_i = null$
  $retval_i := memory(x)$

  comp-write$_i$ (x, v)
  Can occur if $req_i =$ "$write$", $reqaddr_i = x$, $reqval_i(x) = v$, and $retval_i = null$
  $memory(x) := v$
  $retval_i :=$ "OK"

(a) Try to prove, as carefully as you can, that the strong caching system labeled state machine in fact implements the centralized memory labeled state machine.

This proof can be done using ad hoc methods, starting from any execution of the caching system and constructing the (possibly infinite) execution of the centralized memory machine. The construction can proceed by structural induction on the definition of the execution of the caching system.

TA note:

Try to show them the Abstraction Theorem and show how the proof could be made very systematic using this theorem.

Give them direction here, depending on what the individual sections and groups need. They should explore the abstraction issues. Interrupt them once or twice to have them present their ideas so far, so that no group gets too lost.

You may want to draw pictures of the two systems at the start, and make sure that everyone knows what the problem is.

Later, stop them at some point and make sure that they all understand the technique of showing that every trace of SC can be produced by some execution of CM that will leave the two states related by some (trivial) abstraction function.

Valid traces, with data as letters and addresses as integers:

q_0 = all memory set to 'a'.

1. $req\text{-}write_1(7, `f')$
2. $ret\text{-}write_1(\text{“}OK'')$
3. $req\text{-}read_1(7)$
4. $ret\text{-}read_1(`f')$

q_0 = all memory set to 'a'.

1. $req\text{-}write_1(9, `x')$
2. $req\text{-}write_2(9, `y')$
3. $ret\text{-}write_2(\text{“}OK'')$
4. $req\text{-}read_2(9)$
5. $ret\text{-}read_2(`y')$
6. $ret\text{-}write_1(\text{“}OK'')$
7. $req\text{-}read_1(9)$
8. $ret\text{-}read_1(`x')$

q_0 = all memory set to 'a'.

1. $req\text{-}write_1(4, `x')$
2. $req\text{-}write_1(4, `y')$
3. $req\text{-}read_1(4)$
4. $ret\text{-}write_1(\text{“}OK'')$
5. $req\text{-}read_1(4)$
6. $ret\text{-}read_2(8)$
7. $ret\text{-}read_1(`x')$
8. $ret\text{-}read_2(`a')$

It is helpful to figure out what sequences of transitions could have created the above traces, for both SC and CM machines. In other words, where are the hidden internal labels? For starters, remember that the SC machine would have to do a *copy* to a cache before doing a *comp-read*, while the CM machine could just do a *comp-read*.

Solutions and Discussion:

To sketch out the proof, we want to show that any trace of SC is a trace of CM. In other words, any execution α of SC has a corresponding execution β in CM that produces the same externally visible (input and output) labels.

$$trace(\alpha) = trace(\beta)$$

We can show this by induction on the length of α (the number of transitions in that execution). We must first strengthen our hypothesis to include a relation between the states of SC and CM – otherwise, we will not have enough to go on to do the induction. This relation is the *abstraction relation*. Try the relation that *memory* in SC is identical to *memory* in CM.

So, take $P(n)$: For any execution α of SC of length n, there is an execution β of CM that has the same trace and ends with all components except $cache_1, cache_2$ in SC identical to those in CM.

$P(0)$: simple. Start out with the memory equal. No transitions means no labels in the trace, a feat easily duplicated by CM.

$P(n) \Rightarrow P(n+1)$: Take an arbitrary execution

$$\alpha \;=\; \alpha_1, \alpha_2, \ldots, \alpha_n \;,\; \alpha_{n+1}$$

α_1 through α_n form an execution of SC of length n. By the inductive hypothesis, this execution has a corresponding execution of CM β of some length that has the same trace as α and leaves the memories identical:

$$\beta \;=\; \beta_1, \beta_2, \ldots, \beta_m \;,\; ?$$

All we have to show now is that for every possible transition $\alpha_n \rightarrow \alpha_{n+1}$ that SC can make, there is a corresponding sequence of transitions of CM $\beta_m + 1 \rightarrow \beta_m + 2 \rightarrow \ldots \beta_k$ that exhibits the same partial trace as the $\alpha_n \rightarrow \alpha_{n+1}$ transition and leaves the memory of CM in the same state as the memory of SC at α_{n+1}.

The possible transitions of SC are:

- **req-read$_\mathbf{i}$**(x)

  The corresponding transition for CM is also a $req\text{-}read_i$. The same input label is added to the trace, and all state elements are updated identically.

- **req-write$_\mathbf{i}$**(x, v)

  The corresponding transition for CM is also a $req\text{-}write_i$. The same input label is added to the trace, and all state elements are updated identically.

- **ret-read$_\mathbf{i}$**(v)

  The corresponding transition for CM is also a $ret\text{-}read_i$. The same output label is added to the trace, and all state elements are updated identically.

- **ret-write$_i$**($\text{“}OK''$)

  The corresponding transition for CM is also a *ret-write*$_i$. The same output label is added to the trace, and all state elements are updated identically.

- **comp-read$_i$**(x)

  The corresponding transition for CM is also a *comp-read*$_i$. No labels are added to the trace in either case. SC reads from a cache, CM reads from memory. As shown in lecture, in a strong caching system the memory will contain the same value as the cache for non-null entries.

- **comp-write$_i$**(x, v)

  The corresponding transition for CM is also a *comp-read*$_i$. No labels are added to the trace in either case. SC dumps the caches, but they are not reflected in our predicate (the abstraction relation part of it).

- **copy$_i$**(x)

  The corresponding transition for CM is to perform no transition at all. Our predicate remains satisfied.

- **drop$_i$**(x)

  The corresponding transition for CM is to perform no transition at all. Our predicate remains satisfied.

As an aside, the *Abstraction Theorem* makes this proof a little clearer. The application is analogous to using the Termination Theorem to show that a state machine always has finite-length executions: it allows you to skip the induction 'wrapper' because it is a general property of these problems.

Abstraction Relation:
A relation R from states of A to states of B is called an $a$bstraction relation from A to B provided that the following conditions hold:

1. (Start condition)
   If q_A is a start state of A, then there exists a start state q_B of B such that $(q_A, q_B) \in R$.

2. (Step condition)
   If q_A and q_B are reachable states of A and B, respectively, $(q_A, q_B) \in R$, and (q_A, ℓ, q'_A) is a step of A, then there is a sequence of (0 or more) steps of B that:
   – starts with q_B,
   – has the same trace as the step of A, and
   – ends with some state q'_B such that $(q'_A, q'_B) \in R$.

Abstraction Theorem:
If there is an abstraction relation from A to B, then A implements B.