Problem Set 5 Solutions

Problems:

Problem 1 Suppose you were about to enter college today and a college loan officer offered you the following deal: \$25,000 at the start of each year for four years to pay for your college tuition and an option of choosing one of the following repayment plans:

Plan A: Wait four years, then repay $\$20,000$ at the start of each year for the next ten years.

Plan B: Wait five years, then repay $\$30,000$ at the start of each year for the next five years.

Suppose the annual interest rate is currently 7% and does not change in the future.

a) Which repayment plan would you choose?

Solution:

The use of the word 'interest' above is slightly awkward. It refers to the additional money you (or the loan officer) would make if your money was just sitting in a bank. However, we can treat it just as we treat inflation: \$1 today will be worth \$1.07 next year, and $\$1.07^2$ the year after.

Given this, set $r = \frac{1}{1.07}$. Then:

$$\begin{aligned}
A &= \sum_{y=0}^{9} 20000 \cdot r^{y+4} \\
&= r^4 \cdot \sum_{y=0}^{9} 20000 \cdot r^y \\
&= 20000r^4 \cdot \sum_{y=0}^{9} r^y \\
&= 20000r^4 \cdot \frac{1-r^{10}}{1-r} \\
&= \$114,666.69
\end{aligned}$$

$$\begin{aligned}
B &= \sum_{y=0}^{4} 30000 \cdot r^{y+5} \\
&= r^5 \cdot \sum_{y=0}^{4} 30000 \cdot r^y \\
&= 30000r^5 \cdot \sum_{y=0}^{4} r^y \\
&= 30000r^5 \cdot \frac{1-r^5}{1-r} \\
&= \$93,840.63
\end{aligned}$$

You should clearly take Plan B. You will be paying back much less in today's dollars.

b) What is the loan officer's effective profit (in today's dollars) on the loan?

Solution:

The value of the money you are given is:

$$\begin{aligned}
Loan &= \sum_{y=0}^{3} 25000 \cdot r^y \\
&= 25000 \cdot \sum_{y=0}^{3} r^y \\
&= 25000 \cdot \frac{1-r^4}{1-r} \\
&= \$90,607.90
\end{aligned}$$

Therefore, the loan officer's profit is effectively $\$3,233$. (Or $\$24,059$ if we are not on the ball).

Problem 2 Consider the following summation:

$$\sum_{k=2}^{n} \frac{1}{k(k-1)}$$

(a) Find a closed form solution for this summation.
Hint: Try using partial fractions to break $\frac{1}{k(k-1)}$ into a difference of two terms.

Solution:
It is easy to find that $\frac{1}{k(k-1)} = \frac{1}{k-1} - \frac{1}{k}$. So we have:

$$\begin{aligned}
\sum_{k=2}^{n} \frac{1}{k(k-1)} &= \sum_{k=2}^{n} \left(\frac{1}{k-1} - \frac{1}{k} \right) \\
&= \sum_{k=2}^{n} \frac{1}{k-1} - \sum_{k=2}^{n} \frac{1}{k} \\
&= \frac{1}{1} + \frac{1}{2} + \ldots + \frac{1}{n-2} + \frac{1}{n-1} \quad - \\
&\quad \left[\frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n-1} + \frac{1}{n} \right]
\end{aligned}$$

So all that does not cancel out is $1 - \frac{1}{n}$.

(b) Use induction to verify your closed form solution (i.e., prove that it is correct).

Solution:
$P(n) := \left[\sum_{k=2}^{n} \frac{1}{k(k-1)} = 1 - \frac{1}{n} \right]$
Base case: $P(2)$ is true because the only term in the summation is $\frac{1}{2(2-1)} = \frac{1}{2} = 1 - \frac{1}{2}$.
Inductive step: Assuming $P(n)$, show $P(n+1)$.

$$\begin{aligned}
\sum_{k=2}^{n+1} \frac{1}{k(k-1)} &= \sum_{k=2}^{n} \frac{1}{k(k-1)} + \frac{1}{(n+1)(n)} \\
&= 1 - \frac{1}{n} + \frac{1}{(n+1)(n)} \qquad \textbf{IH} \\
&= 1 - \frac{n+1}{(n+1)n} + \frac{1}{(n+1)(n)} \\
&= 1 - \frac{n}{(n+1)n} \\
&= 1 - \frac{1}{n+1}
\end{aligned}$$

Problem 3 Find closed-form expressions for the following sums:

(a)

$$\sum_{k=0}^{n} \frac{8^k - 5^k}{9^k}$$

Solution:

$$\begin{aligned}
\sum_{k=0}^{n} \frac{8^k - 5^k}{9^k} &= \sum_{k=0}^{n} \left[\frac{8^k}{9^k} - \frac{5^k}{9^k} \right] \\
&= \sum_{k=0}^{n} \left[\left(\frac{8}{9}\right)^k - \left(\frac{5}{9}\right)^k \right] \\
&= \sum_{k=0}^{n} \left(\frac{8}{9}\right)^k - \sum_{k=0}^{n} \left(\frac{5}{9}\right)^k \\
&= \frac{1 - \left(\frac{8}{9}\right)^{n+1}}{1 - \frac{8}{9}} - \frac{1 - \left(\frac{5}{9}\right)^{n+1}}{1 - \frac{5}{9}} \\
&= \frac{1 + \frac{1}{3}\left(\frac{5}{9}\right)^{n+1} - \left(\frac{8}{9}\right)^{n+1}}{12}
\end{aligned}$$

(b)

$$\sum_{i=1}^{n} \sum_{j=0}^{\infty} i^{7/16} \cdot \left(1 - \frac{1}{4i^{9/16}}\right)^j$$

Solution:

This sum is actually just a sum of a geometric series followed by the sum of an arithmetic series.

$$\begin{aligned}
\sum_{i=1}^{n} \sum_{j=0}^{\infty} i^{7/16} \cdot \left(1 - \frac{1}{4i^{9/16}}\right)^j &= \sum_{i=1}^{n} i^{7/16} \cdot \frac{1}{1 - \left(1 - \frac{1}{4i^{9/16}}\right)} \\
&= \sum_{i=1}^{n} 4i \\
&= 2n(n+1)
\end{aligned}$$

Problem 4 *Note: This problem is only worth one point. Give it some thought but don't exhaust yourself on it.*

Consider the sum of the following sequence:

$$1 - 1 + \frac{1}{2} - \frac{1}{2} + \frac{1}{3} - \frac{1}{3} + \frac{1}{4} - \frac{1}{4} + \frac{1}{5} - \frac{1}{5} + \ldots$$

To sum it, suppose we rearrange the terms and write it as:

$$\begin{array}{cc}
(1 + \frac{1}{2} - 1) & + \\
(\frac{1}{3} + \frac{1}{4} - \frac{1}{2}) & + \\
(\frac{1}{5} + \frac{1}{6} - \frac{1}{3}) & + \\
(\frac{1}{7} + \frac{1}{8} - \frac{1}{4}) & + \\
\ldots &
\end{array}$$

which we can write as a summation as:

$$a = \sum_{n=1}^{\infty} \left(\frac{1}{2n-1} + \frac{1}{2n} - \frac{1}{n} \right) = \sum_{n=1}^{\infty} \frac{1}{2n(2n-1)}$$

On the other hand, we arrange the original summation as

$$\begin{array}{cc}
(1 - 1 - \frac{1}{2}) & + \\
(\frac{1}{2} - \frac{1}{3} - \frac{1}{4}) & + \\
(\frac{1}{3} - \frac{1}{5} - \frac{1}{6}) & + \\
(\frac{1}{4} - \frac{1}{7} - \frac{1}{8}) & + \\
\cdots &
\end{array}$$

$$b = \sum_{n=1}^{\infty} \left(\frac{1}{n} - \frac{1}{2n-1} - \frac{1}{2n} \right) = \sum_{n=1}^{\infty} \frac{-1}{2n(2n-1)} = -a$$

How is this possible? Choose the correct response from the options below. Be prepared to back up your answer.

a) I find the above somewhat disturbing, but I can't resolve it.

b) Don't toy with me. Your petty tricks are no match for my superior mathematical skills.

c) It's 4am Tuesday morning. I'm not even coherent enough to be disturbed.

Solution:

Of course, this is a personal question. However, if you answered **c**, you really should get started earlier – for your own good.

Students who answered **b** may have seen a hint it the two harmonic summations (positive and negative) hiding within the original series. The original summation is only *conditionally* convergent. (This means that if you sum the absolute value of the terms, $2 \cdot \sum_{n=1}^{\infty} \frac{1}{n}$, the summation is not finite). With such summations, you cannot freely rearrange the terms. In fact, there is a theorem that says that any series that converges but does not converge *absolutely* can be rearranged to form any real number.

It is not insightful to declare that $a = 0 = -0$. While this is part of the contradiction above, it is clearly not the answer. Every term in the first summation is positive. Every term in the second summation is negative. Neither can be zero, as a result. In fact, they are somewhere around $\pm log(2)$. Similarly, it is not adequate to say that the sums diverge or are not well-defined. Both summations, as well as the original zero-sum, really do converge on finite numbers.

Don't worry about the details here. However, you should take from this a warning about being too loose with your math, much as you must avoid making flawed but convincing proofs.

Problem 5 The following labeled state machine *DSet* models a "dynamic set" of integers. (Remember that a set, unlike a multiset, can only contain one element of any particular object). The set starts out empty.

A client can modify or inspect the set by means of the following operations:

- **add**(x), which adds an integer x to the set.
- **remove**(x), which removes integer x from the set.
- **isin**(x), which tells if the element x is in the set.

Formally, *DSet* can be defined as a collection of the following components:

- Q: The states consist of the components:
  - *set*, a set of integers
  - $reqtype \in \{\text{“}add\text{”}, \text{“}remove\text{”}, \text{“}isin\text{”}\} \cup \{null\}$
  - $element \in \mathbb{Z} \cup \{null\}$
  - $retval \in \{\text{“}OK\text{”}, null\} \cup Boolean$
- Q_0: Initially, $set = \emptyset$, and all the other components are *null*. (Note that *null* is a special symbol, not the empty set $\emptyset$).
- L:

  Input labels:

  request-add(x), $x \in \mathbb{Z}$
  request-remove(x), $x \in \mathbb{Z}$
  request-isin(x), $x \in \mathbb{Z}$

  Internal labels:

  compute-add(x), $x \in \mathbb{Z}$
  compute-remove(x), $x \in \mathbb{Z}$
  compute-isin(x), $x \in \mathbb{Z}$

  Output labels:

  return$(\text{“}OK\text{”})$
  return(b), $b \in Boolean$

- Transitions (δ):

  request-add(x)
  if $reqtype = null$ then
  $reqtype := \text{“}add\text{”}$
  $element := x$

  request-remove(x)
  if $reqtype = null$ then
  $reqtype := \text{“}remove\text{”}$
  $element := x$

request-isin(x)
if $reqtype = null$ then
  $reqtype :=$ "$isin$"
  $element := x$

compute-add(x)
Can occur if $reqtype =$ "add", $element = x$, and $retval = null$
  $set := set \cup \{x\}$
  $retval :=$ "OK"

compute-remove(x)
Can occur if $reqtype =$ "$remove$", $element = x$, and $retval = null$
  $set := set - \{x\}$
  $retval :=$ "OK"

compute-isin(x)
Can occur if $reqtype =$ "$isin$", $element = x$, and $retval = null$
if $x \in set$ then
  $retval := true$
else
  $retval := false$

return("OK")
Can occur if $retval =$ "OK"
  $reqtype, element, retval := null$

return(b), $b \in Boolean$
Can occur if $retval = b$
  $reqtype, element, retval := null$

(a) Which of the following are valid traces of *DSet*? For any incorrect ones, indicate what is wrong.

1. Start with the sole element of Q_0.
   - **request-add**(7)
   - **return**("OK")
   - **request-isin**(7)
   - **return**($true$)
   - **request-remove**(7)
   - **return**("OK")
   - **request-remove**(7)
   - **return**("OK")
2. Start with the sole element of Q_0.
   - **request-add**(7)
   - **return**("OK")
   - **request-remove**(4)

- **return**(*false*)
- **request-add**(7)
- **return**("*OK*")

3. Start with the sole element of Q_0.

   - **request-add**(4)
   - **request-add**(6)
   - **return**("*OK*")
   - **request-isin**(6)
   - **request-isin**(4)
   - **return**(*true*)

4. Start with the sole element of Q_0.

   - **request-add**(3)
   - **request-remove**(3)
   - **return**("*OK*")
   - **request-isin**(3)
   - **return**(*false*)
   - **request-remove**(1)
   - **return**("*OK*")

Solution:

1. valid
2. invalid – remove computations always set the return value to "*OK*", whether or not the element was really removed.
3. invalid – the second add request would have been ignored, so the check for 6 should return false.
4. invalid – similarly, the first remove request would have been ignored, since the add request is still 'pending' (*reqtype* is not *null*).

Note that it is not enough to imagine one execution that would not produce such a trace. A trace is possible if *any* execution allowed by the rules of the transitions would produce that trace.

(b) High-level implementation:

One way to implement *DSet* is to represent the set by a finite sequence of integers. The **add** operation could append the new element to the end of the sequence *whether or not it already appears in the sequence.* The **isin** operation returns the answer to whether or not any copy of the element is in the sequence. The **remove** operation removes the element from the sequence at every place that it occurs.

Define a state machine *Seq* corresponding to this implementation. In implementing the operations, you should not try to represent individual search steps that try to locate an element in the sequence. Instead, let each operation perform its main computation in one large-granularity step, which involves the entire sequence. Thus, this description will still be rather abstract.

Solution:

- Q: The states consist of the components:
  - seq, a sequence (ordered) of integers
  - $reqtype \in \{\text{"}add\text{"}, \text{"}remove\text{"}, \text{"}isin\text{"}\} \cup \{null\}$
  - $element \in \mathbb{Z} \cup \{null\}$
  - $retval \in \{\text{"}OK\text{"}, null\} \cup Boolean$
- Q_0: Initially, seq has no elements, and all the other components are $null$.
- L: identical to abstract case.
- δ: Input, output are identical to abstract case.

compute-add(x)
Can occur if $reqtype = \text{"}add\text{"}$, $element = x$, and $retval = null$
  $seq := seq$ with x appended on the end
  $retval := \text{"}OK\text{"}$

compute-remove(x)
Can occur if $reqtype = \text{"}remove\text{"}$, $element = x$, and $retval = null$
  $seq := seq$ compressed to not include any elements equal to x
  $retval := \text{"}OK\text{"}$

compute-isin(x)
Can occur if $reqtype = \text{"}isin\text{"}$, $element = x$, and $retval = null$
if $x \in seq$ then
  $retval := true$
else
  $retval := false$

(c) Prove that your *Seq* state machine implements the *DSet* labeled state machine; that is, prove that any trace of *Seq* is a trace of *DSet*. Remember that a trace is a listing of only the input and output labels during an execution – the externally visible behavior. You may use the Abstraction Theorem.

Solution:

We want to show that any trace of Seq is a trace of $DSet$. In other words, any execution α of Seq has a corresponding execution β in $DSet$ that produces the same externally visible (input and output) labels.

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

We 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 Seq and $DSet$ – otherwise, we will not have enough to go on to do the induction. This relation is the *abstraction relation*.

Use the relation that *reqtype*, *element*, and *retval* are identical, and *seq* contains the same elements as *set* (although they may be repeated in *seq*).

So, our proposition would be $P(n)$: For any execution α of Seq of length n, there is an execution β of $DSet$ that has the same trace and ends with *reqtype*, *element*, and *retval* in $DSet$ identical to those in Seq and with the elements of *seq* and *set* being identical.

Given the Abstraction Theorem, we only have to show that these conditions are preserved for any transition that Seq can make.

The possible transitions of Seq are:

request-add(x)
Also do a *request-add*(x). Nothing changes besides the identical alterations, and the labels generated are the same. (By the abstraction relation, we know that the *reqtype* in one execution will be *null* if and only if it is *null* in the other).

request-remove(x)
Also do a *request-remove*(x). Nothing changes besides the identical alterations, and the labels generated are the same.

request-isin(x)
Also do a *request-isin*(x). Nothing changes besides the identical alterations, and the labels generated are the same.

compute-add(x)
Do a *compute-add*(x). While *seq* may now have several copies of x, this is allowed by our abstraction relation. Otherwise, the conditions are maintained.

compute-remove(x)
Do a *compute-add*(x). While *seq* may have had several copies of x, all are removed by the Seq transition.

compute-isin(x)
Do a *compute-isin*(x). Will find the same result in both *seq* and *set*, because the abstraction relation says that they have the same elements.

return$(\text{“}OK\text{”})$
Also do a *return*$(\text{“}OK\text{”})$. Nothing changes besides the identical alterations, and the labels generated are the same.

return(b), $b \in Boolean$
Also do a *return*(b). Nothing changes besides the identical alterations, and the labels generated are the same.

(d) Low-level implementation:

Give an implementation called *SeqLow* that relies on lower-level operations for performing the searches and removals (operate at the element-level).

Solution:

- Q: The states consist of the components:
  - seq, a sequence (ordered) of integers seq_0 through seq_n
  - $index \in \mathbb{N}$
  - $offset \in \mathbb{N}$
  - $maxindex \in \mathbb{N}$
  - $reqtype \in \{\text{"}add\text{"}, \text{"}remove\text{"}, \text{"}isin\text{"}\} \cup \{null\}$
  - $element \in \mathbb{Z} \cup \{null\}$
  - $retval \in \{\text{"}OK\text{"}, null\} \cup Boolean$
- Q_0: Initially, seq has no elements, $index = 0$, $maxindex = 0$, $offset = 0$, and all the other components are $null$.
- L: Input, output identical to abstract case.

  Internal labels:

  compute-add(x), $x \in \mathbb{Z}$
  compute-remove(x), $x \in \mathbb{Z}$
  compute-isin(x), $x \in \mathbb{Z}$
- δ: Input, output are identical to abstract case.

  compute-add(x)
  Can occur if $reqtype = \text{"}add\text{"}$, $element = x$, and $retval = null$
  $seq_{index} := x$
  $maxindex := maxindex + 1$
  $retval := \text{"}OK\text{"}$

  compute-remove(x)
  Can occur if $reqtype = \text{"}remove\text{"}$, $element = x$, and $retval = null$
  if $index = maxindex$ then
  $index := 0$
  $maxindex := maxindex - offset$
  $retval := \text{"}OK\text{"}$
  else if $seq_{index} = x$ then
  $index := index + 1$
  $offset := offset + 1$
  else
  $seq_{index\text{-}offset} := seq_{index}$
  $index := index + 1$

compute-isin(x)
Can occur if $reqtype$ = "$isin$", $element = x$, and $retval = null$
if $index = maxindex$ then
  $index := 0$
  $retval := false$
else if $seq_{index} = x$ then
  $index := 0$
  $retval := true$
else
  $index := index + 1$

(e) Give an informal argument that *SeqLow* implements *Seq*.

Solution:

We must show that every sequence of *SeqLow* is a sequence of *Seq*. Informally, we can argue that for any execution A of *SeqLow* that has the same trace as an execution B of *Seq*, every transition from A can be matched by some sequence of transitions from B. We can match up the components of the two systems very closely.

We take as an invariant that all like-named components in the two systems have the same values. $maxindex$ is one more than the largest index in the sequence. $index$ holds the current place in the sequence that we are examining. If $reqtype$ = "$isin$", then we know that there are no elements equal to $element$ from seq_0 to $seq_{index-1}$. If $reqtype$ = "$remove$", then we know that $offset$ is equal to the number of elements that were equal to $element$ from seq_0 to $seq_{index-1}$, and also that all of those elements have been shifted down to fill in. (It is actually rather difficult to express these constraints more formally).

Input and output transitions are exactly matched.

compute-add(x)
Also do a *compute-add*(x). Will add the element to the end in both cases.

compute-remove(x)
if $index = maxindex$ then
Also do a *compute-remove*(x). In both cases, all elements that match x are gone from the sequence, and the sequence is that much shorter, else if $seq_{index} = x$ then
Do nothing. No observable changes; just stepping through details. else
Do nothing. No observable changes; just stepping through details.

compute-isin(x)
if $index = maxindex$ then
Also do a *compute-isin*(x). Will return the same value, since we now know that x is not equal to any seq_i where $i < maxindex$. else if $seq_{index} = x$ then
Also do a *compute-isin*(x). Will return the same value, since we know that x is equal to some seq_i (namely, $i = index$). else
Do nothing. No observable changes; just stepping through details.

Problem 6 Consider an "abstract object" that maintains a dynamic polynomial *poly*. This polynomial has variable x, and integer coefficients. Initially, the polynomial is just the 0 polynomial (no terms). The polynomial supports operations:

- **add-term**(c, e), which adds the term cx^e to the maintained polynomial.
- **coeff**(e), which asks for the coefficient of the x^e term.

(a) Model this object by a labeled state machine ADP. The state should keep track of the polynomial (a value that is simply a mathematical polynomial – you don't have to think about how to represent it using lower-level mathematical objects like numbers, matrices, sequences, etc.) The state will also need some bookkeeping information to keep track of operations in progress. The labels should include inputs reflecting the requests and outputs reflecting the responses. Your machine should ignore requests that arrive while an operation is active.

Solution:

- Q:
  $p \in \{polynomials\}$
  $req \in \{\text{"}add\text{-}term\text{"}, \text{"}find\text{-}coeff\text{"}, null\} \cup \{\text{"}OK\text{"}\} \cup \mathbb{Z}$
  $power \in \mathbb{N} \cup \{null\}$
  $mult \in \mathbb{Z} \cup \{null\}$
- Q_0:
  A set with only one element:
  $\{(0, null, null, null)\}$
  or, equivalently,
  $p = 0$, $req = null$, $power = null$, $mult = null$
- L:

  Input labels:
  $req\text{-}add\text{-}term(c, e), \quad c \in \mathbb{Z}, e \in \mathbb{N}$
  $req\text{-}coeff(e), \quad e \in \mathbb{N}$
  Internal labels:
  $comp\text{-}sum(c, e), \quad c \in \mathbb{Z}, e \in \mathbb{N}$
  $comp\text{-}coeff(e), \quad e \in \mathbb{N}$
  Output labels:
  $return(\text{"}OK\text{"})$
  $return(c), \quad c \in \mathbb{Z}$
- δ:
  Each is associated with (generates) the label of its namesake.

  1. $req\text{-}add\text{-}term(c, e)$:
     Can occur anytime.
     if $req = null$ then

$req :=$ "*add*"
$power := e$
$mult := c$

2. $req\text{-}coeff(e)$:
   Can occur anytime.
   if $req = null$ then
   $req :=$ "$find\text{-}coeff$"
   $power := e$
   $mult := null$

3. $comp\text{-}sum(c, e)$:
   Can occur if $req =$ "*add-term*", $mult = c$, and $power = e$.
   $p := p + c \cdot x^e$
   $req :=$ "OK"

4. $comp\text{-}coeff(e)$:
   Can occur if $req =$ "$find\text{-}coeff$" and $power = e$.
   $req :=$ coefficient of x^e in p

5. $return(\text{"}OK\text{"})$:
   Can occur if $req =$ "OK".
   $req, power, mult := null$

6. $return(c)$:
   Can occur if $req = c$.
   $req, power, mult := null$

(b) List some (4–5) traces that are exhibited by your state machine ADP. Include at least one that involves a request arriving when it is not supposed to.

Solution:

Start with $(0, null, null, null)$:

$$
\begin{array}{l}
req\text{-}add\text{-}term(1,1) \\
return(\text{"}OK\text{"}) \\
req\text{-}add\text{-}term(2,2) \\
return(\text{"}OK\text{"}) \\
req\text{-}coeff(2) \\
return(2) \\
req\text{-}add\text{-}term(2,1) \\
return(\text{"}OK\text{"}) \\
req\text{-}coeff(1) \\
return(3)
\end{array}
$$

Start with $(0, null, null, null)$:

$req\text{-}add\text{-}term(1, 1)$
$return(\text{“}OK\text{”})$
$req\text{-}add\text{-}term(2, 2)$
$req\text{-}coeff(2)$
$return(\text{“}OK\text{”})$
$req\text{-}coeff(2)$
$req\text{-}add\text{-}term(3, 2)$
$return(2)$
$req\text{-}coeff(2)$
$return(2)$

Start with $(0, null, null, null)$:

$req\text{-}add\text{-}term(4, 2)$
$req\text{-}add\text{-}term(7, 1)$
$return(\text{“}OK\text{”})$
$req\text{-}coeff(3)$
$req\text{-}coeff(2)$
$return(0)$
$req\text{-}coeff(1)$
$return(0)$
$req\text{-}coeff(2)$
$return(4)$

Of course, there are many more traces. The important thing about this system is that requests will be dropped if there is still a pending request that has not generated a *return* – the value of *req* will not be *null* when they come in.

(c) Describe in words an implementation of your state machine *ADP*. The state of this implementation should involve lower-level mathematical objects like numbers, matrices, sequences, etc.

Solution:

Using Arrays: Keep an array *poly* such that the ith entry holds the coefficient of x^i in p. The array size will have to grow as necessary, or there will be a limit on the terms of the polynomial. $comp\text{-}add\text{-}term(c, e)$ will simply be $poly[e] := poly[e] + c$. $comp\text{-}coeff(e)$ will be $req := poly[e]$.

Using Ordered Pairs: Keep a set *terms* of ordered pairs (i, c) mapping the ith entry to the coefficient of x^i in p. $comp\text{-}add\text{-}term(c, e)$ will be more complex, unless we just add the pair and sum all pairs with like first components on lookup. We will have to search through for an existing pair with a matching first component. This will require more internal state. $comp\text{-}coeff(e)$ will be a similar search for the right pair, returning 0 if none is found.

(d) Formalize your implementation as a labeled state machine *Imp*. It should have the same external labels as your labeled state machine *ADP*.

Solution:

Using Arrays: Assume that we ignore the bounds of the array.

We need the following modifications to the abstract machine:

- Q:
  For all (many) $i \in \mathbb{N}$,
  $poly[i] \in \mathbb{Z}$
  $req \in \{\text{"}add\text{-}term\text{"}, \text{"}find\text{-}coeff\text{"}, null\} \cup \{\text{"}OK\text{"}\} \cup \mathbb{Z}$
  $power \in \mathbb{N} \cup \{null\}$
  $mult \in \mathbb{Z} \cup \{null\}$
- Q_0:
  A set with only one element:
  $poly[i] = 0$for all i, $req = null$, $power = null$, $mult = null$
- L: Same as before.
- δ: Same as before for input and output.
  - $comp\text{-}sum(c, e)$:
    Can occur if $req = \text{"}add\text{-}term\text{"}$, $mult = c$, and $power = e$.
    $poly[e] := poly[e] + c$
    $req := \text{"}OK\text{"}$
  - $comp\text{-}coeff(e)$:
    Can occur if $req = \text{"}find\text{-}coeff\text{"}$ and $power = e$.
    $req := poly[e]$

Using Unique Ordered Pairs: We need the following modifications to the abstract machine:

- Q:
  A linked list $terms$ of the form $(i, m, next)$ with $i \in \mathbb{N}$, $m \in \mathbb{Z}$, $next \in terms \cup \{null\}$.
  $firstterm \in terms$
  $curterm \in terms \cup \{null\}$
  $req \in \{\text{"}add\text{-}term\text{"}, \text{"}find\text{-}coeff\text{"}, null\} \cup \{\text{"}OK\text{"}\} \cup \mathbb{Z}$
  $power \in \mathbb{N} \cup \{null\}$
  $mult \in \mathbb{Z} \cup \{null\}$
- Q_0:
  A set with only one element:
  $firstterm = (0, 0, null)$, $curterm = firstterm$, $terms = \{firstterm\}$, $req = null$, $power = null$, $mult = null$
- L: Same as before for input and output labels.

  Internal labels:

  $comp\text{-}sum(c, e), \quad c \in \mathbb{Z}, e \in \mathbb{N}$
  $comp\text{-}sum\text{-}next(c, e), \quad c \in \mathbb{Z}, e \in \mathbb{N}$

$comp\text{-}sum\text{-}default(c, e), \quad c \in \mathbb{Z}, e \in \mathbb{N}$
$comp\text{-}coeff(e), \quad e \in \mathbb{N}$
$comp\text{-}coeff\text{-}next(e), \quad e \in \mathbb{N}$
$comp\text{-}coeff\text{-}default(e), \quad e \in \mathbb{N}$

- δ: Input, output transitions same as before.
  - $comp\text{-}sum(c, e)$:
    Can occur if $req =$ "$add\text{-}term$", $mult = c$, $power = e$, $curterm \neq null$, $curterm.i = e$
    $curterm.m := curterm.m + c$
    $req :=$ "OK"
    $curterm := firstterm$
  - $comp\text{-}sum\text{-}next(c, e)$:
    Can occur if $req =$ "$add\text{-}term$", $mult = c$, $power = e$, $curterm \neq null$, $curterm.i \neq e$
    $curterm := curterm.next$
  - $comp\text{-}sum\text{-}default(c, e)$:
    Can occur if $req =$ "$add\text{-}term$", $mult = c$, $power = e$, $curterm = null$
    $firstterm := (e, c, firstterm)$
    $req :=$ "OK"
    $curterm := firstterm$
  - $comp\text{-}coeff(e)$:
    Can occur if $req =$ "$find\text{-}coeff$", $power = e$, $curterm \neq null$, $curterm.i = e$
    $req := curterm.m$
    $curterm := firstterm$
  - $comp\text{-}coeff\text{-}next(e)$:
    Can occur if $req =$ "$find\text{-}coeff$", $power = e$, $curterm \neq null$, $curterm.i \neq e$
    $curterm := curterm.next$
  - $comp\text{-}coeff\text{-}default(e)$:
    Can occur if $req =$ "$find\text{-}coeff$", $power = e$, $curterm = null$
    $req := 0$
    $curterm := firstterm$

Note that we implicitly added a new term to *terms* every time we expanded the linked list.

Using Multiple Ordered Pairs: We need the following modifications to the abstract machine:

- Q:
  A linked list *terms* of the form $(i, m, next)$ with $i \in \mathbb{N}$, $m \in \mathbb{Z}$, $next \in terms \cup \{null\}$.
  $firstterm \in terms$
  $curterm \in terms \cup \{null\}$
  $total \in \mathbb{Z}$
  $req \in \{$"$add\text{-}term$", "$find\text{-}coeff$", $null\} \cup \{$"OK"$\} \cup \mathbb{Z}$
  $power \in \mathbb{N} \cup \{null\}$
  $mult \in \mathbb{Z} \cup \{null\}$

- Q_0:
  A set with only one element:
  $firstterm = (0, 0, null)$, $curterm = firstterm$, $terms = \{firstterm\}$, $total = 0$, $req = null$, $power = null$, $mult = null$
- L: Same as before for input and output labels.

  Internal labels:

  $comp\text{-}sum(c, e), \quad c \in \mathbb{Z}, e \in \mathbb{N}$
  $comp\text{-}coeff(e), \quad e \in \mathbb{N}$
  $comp\text{-}coeff\text{-}next(e), \quad e \in \mathbb{N}$
  $comp\text{-}coeff\text{-}end(e), \quad e \in \mathbb{N}$
- δ: Input, output transitions same as before.
  - $comp\text{-}sum(c, e)$:
    Can occur if $req =$ "*add-term*", $mult = c$, and $power = e$
    $firstterm := (e, c, curterm)$
    $req :=$ "OK"
  - $comp\text{-}coeff(e)$:
    Can occur if $req =$ "$find\text{-}coeff$", $power = e$, $curterm \neq null$, and $curterm.i = e$
    $total := total + curterm.m$
    $curterm := curterm.next$
  - $comp\text{-}coeff\text{-}next(e)$:
    Can occur if $req =$ "$find\text{-}coeff$", $power = e$, $curterm \neq null$, and $curterm.i \neq e$
    $curterm := curterm.next$
  - $comp\text{-}coeff\text{-}end(e)$:
    Can occur if $req =$ "$find\text{-}coeff$", $power = e$, and $curterm = null$
    $req := total$
    $curterm := firstterm$
    $total := 0$

(e) Argue that your implementation *Imp* is correct, in the sense that every trace of *Imp* is also a trace of ADP.

Solution:

Using Arrays: This is the easiest.

The abstraction function is obvious: All corresponding elements have the same values and for every term cx^i in p, $poly[i] = c$. Only the to differing transitions are nontrivial to examine. (Otherwise, the executions are identical and the abstraction function is clearly preserved). However, even *comp-sum* and $comp\text{-}coeff$ are easily accounted for: if an execution of Imp is extended by either, extend an execution of ADP with the same trace using the same transition. *comp-sum* preserves the abstraction function by performing matching arithmetic. $comp\text{-}coeff$ preserves the abstraction function because p and *poly* are in correspondence and will set req to the same value.

Using Unique Pairs: Somewhat tricky to do formally.

The abstraction function is: elements of the same name have the same values; every non-zero term in p has exactly one corresponding triplet in $terms$, and every zero-coefficient term in p either has no corresponding triplet or has a triplet with a second component of 0.

We need other elements of an invariant: In Imp, $firstterm$ begins a linked list that reaches all terms in $terms$; there are no terms with $i = mult$ 'between' $firstterm$ and $curterm$; no two terms have the same i component; if req is not "$find\text{-}coeff$" or "$add\text{-}term$" then $curterm = firstterm$.

Input and output transitions have a simple identity correspondence. Roughly, if the last transition of the partial execution of Imp is:

comp-sum$(\mathbf{c}, \mathbf{e})$
perform a $comp\text{-}sum(c, e)$ transition. Both set req to "OK"; the assignments keep the polynomial structures aligned. (We use the fact that the IH tells us that no other element of $terms$ represents the same exponent here). Although req is changed, $curterm$ is set to $firstterm$. Nothing else is changed.

comp-sum-next$(\mathbf{c}, \mathbf{e})$
do nothing. No data is changed, and the invariant holds over the change in $curterm$ because $i \neq mult$ for the old $curterm$.

comp-sum-default$(\mathbf{c}, \mathbf{e})$
perform a $comp\text{-}sum(c, e)$ transition. If there are no terms after $curterm$, and no terms with $i = mult$ between $firstterm$ and $curterm$, then there must be no terms at all with $i = mult$. So adding in the new term did not violate uniqueness. The new $firstterm$ is connected to all of the elements of $terms$ because the old $firstterm$ was.

comp-coeff$(\mathbf{e})$
perform a $comp\text{-}coeff(e)$ transition. Must be the correct coefficient because p and $terms$ are in correspondence and no two terms have the same i component.

comp-coeff-next$(\mathbf{e})$
do nothing. No data is changed, and the invariant holds over the change in $curterm$ because $i \neq mult$ for the old $curterm$.

comp-coeff-default$(\mathbf{e})$
perform a $comp\text{-}coeff(e)$ transition. If there are no terms after $curterm$, and no terms with $i = mult$ between $firstterm$ and $curterm$, then there must be no terms at all with $i = mult$. By the correspondence between p and $terms$, the coefficient must be 0 in both ADP and Imp.

Although the steps are not filled in, the above is a basic reasoning that traces of Imp can be matched by traces of ADP.

Using Multiple Pairs: Slightly trickier to do formally.

The abstraction function is: elements of the same name have the same values; every term in p has a coefficient equal to the sum of m in all triplets $(i, m, next)$ in $terms$ with i equal to that coefficient.

We need other elements of an invariant: In Imp, $firstterm$ begins a linked list that reaches all terms in $terms$; $total$ is the sum of m in all triplets $(i, m, next)$ between $firstterm$ and $curterm$; if req is not "$find\text{-}coeff$" then $curterm = firstterm$.

Input and output transitions have a simple identity correspondence. Roughly, if the last transition of the partial execution of *Imp* is:

comp-sum(c, e)
perform a *comp-sum*(c, e) transition. Both set *req* to "*OK*". The assignments keep the polynomial structures aligned by adding in the new coefficient. The new *firstterm* is connected to all of the elements of *terms* because the old *firstterm* was. Nothing else is changed.

comp-coeff(e)
do nothing. *total* is appropriately increased, since *curterm* is advanced beyond the term that is contributing.

comp-coeff-next(e)
do nothing. No data is changed, and the invariant holds over the change in *curterm* because $i \neq mult$ for the old *curterm*.

comp-coeff-end(e)
perform a *comp-coeff*(e) transition. If there are no terms after *curterm*, then all terms with $i = mult$ are between *firstterm* and *curterm*. Thus, *total* must be the proper coefficient of term e. Since p and *terms* are in correspondence, *Imp* and *ADP* will update *req* to the same value.

Problem 7 Attached to this problem set are some students' solutions from a previous problem set. Mark up these solutions, indicating things that are incorrect or unclear, and if possible show how the solution can be made completely correct and clear. Please refer to the 6.042 course webpage for more readable versions of the students' solutions.