15-150 Spring 2012
Homework 03

Out: Tuesday, 31 January 2012
Due: Wednesday, 8 February 2012 at 09:00 EST

1 Introduction

This homework will focus on writing functions on lists and proving properties of them. This homework is longer and harder than the previous two: start early!

1.1 Getting The Homework Assignment

The starter files for the homework assignment have been distributed through our `git` repository as usual.

1.2 Submitting The Homework Assignment

To submit your solutions, place your `hw03.pdf` and modified `hw03.sml` files in your handin directory on AFS:

```
/afs/andrew.cmu.edu/course/15/150/handin/<yourandrewid>/hw03/
```

Your files must be named exactly `hw03.pdf` and `hw03.sml`. After you place your files in this directory, run the check script located at

```
/afs/andrew.cmu.edu/course/15/150/bin/check/03/check.pl
```

then fix any and all errors it reports.

Remember that the check script is *not* a grading script—a timely submission that passes the check script will be graded, but will not necessarily receive full credit.

Also remember that your written solutions must be submitted in PDF format—we do not accept MS Word files.

Your `hw03.sml` file must contain all the code that you want to have graded for this assignment and compile cleanly. If you have a function that happens to be named the same as one of the required functions but does not have the required type, it will not be graded.

1.3 Methodology

You must use the five step methodology for writing functions for every function you write on this assignment. In particular, you will lose points for omitting the purpose, examples, or tests even if the implementation of the function is correct.

1.4 Due Date

This assignment is due on Wednesday, 8 February 2012 at 09:00 EST. Remember that this deadline is final and that we do not accept late submissions.

2 Zippidy Doo Da

It's often convenient to take a pair of lists and make one list of pairs from it. For instance, if we have the lists

$$[5, 1, 2, 1] \qquad \text{and} \qquad [\text{“}a\text{”}, \text{“}b\text{”}]$$

we might be interested in the list

$$[(5, \text{“}a\text{”}), (1, \text{“}b\text{”})]$$

Task 2.1 (5%). Write the function

```
zip : int list * string list -> (int * string) list
```

that performs the transformation of pairing the n^{th} element from the first list with the n^{th} element of the second list. If your function is applied to a pair of lists of different length, the length of the returned list should be the minimum of the lengths of the argument lists. You should ensure that `zip` is a total function (but you do not need to formally prove this fact).

Task 2.2 (5%). Write the function

```
unzip : (int * string) list -> int list * string list
```

`unzip` does the opposite of `zip` in the sense that it takes a list of tuples and returns a tuple of lists, where the first list in the tuple is the list of first elements and the second list is the list of second elements. You should ensure that `unzip` is a total function (but you do not need to formally prove this fact).

Task 2.3 (10%). Prove Theorem 1.

Theorem 1. *For all* `l : (int * string) list`, *`zip(unzip l)`* $\cong$ *l.*

Be sure to use the template for a proof by *structural induction on lists*; see the Lecture 4 notes. In your proof, be sure to state when you are using valuability, and explain why the expressions in question are valuable. You may use totality of `zip` and `unzip` in your explanation but need to cite such uses carefully.

Solution 2.3

Proof. This proof will be by structural induction on l.

Case for `[]`:
To Show: `zip (unzip [])` $\cong$ `[]`.

First we evaluate `unzip []`:

	`unzip []`	
$\cong$	`case [] of [] => ([],[]) | ...`	step
$\cong$	`([],[])`	step

We now evaluate `zip([],[])`:

```
  zip([],[])
≅ case ([],[]) of ([],_) => [] | (_,[]) => [] | ...        step
≅ []                                                        step
```

We have shown that `zip (unzip [])` ≅ `[]`. This completes the case.

Case for `x::xs`:
Inductive Hypothesis: `zip (unzip xs)` ≅ `xs`

To show: `zip (unzip x::xs)` ≅ `x::xs`. Because x is a value of type `int * string`, `x` ≅ `(i,j)` for some integer i and string j.

Thus, we can calulate as follows:

```
  unzip x::xs
≅ case (i,j)::xs of ...
    (x1,x2)::xs => let val (l1,l2) = unzip xs ...        step
≅ let val (l1,l2) = unzip xs in (i::l1,j::l2) end        step
```

By the totality of `unzip`, we know that `unzip l` is valuable, so `unzip l` ≅ `(l1',l2')` for some two lists l_1' and l_2'. Using this, we continue evaluating the ML:

```
≅ let val (l1,l2) = (l1',l2') in (i::l1,j::l2) end      step
≅ (i::l1',j::l2')                                        step
```

We have now shown that `unzip x::xs` ≅ `(i::l1',j::l2')`. We now plug these lists into `zip` to get:

```
  zip (unzip x::xs)
≅ zip (i::l1',j::l2')                              substitution
≅ case (i::l1',j::l2') of ...
    (x::xs,y::ys) => (x,y) ::  zip(xs,ys)                  step
≅ (i,j) ::  zip(l1',l2')                                   step
≅ (i,j) ::  zip(unzip xs)                          substitution
≅ (i,j) ::  xs                                            By IH
≅ x ::  xs                                         substitution
```

Thus, we have established that `zip (unzip x::xs)` $\cong$ `x::xs`. This finishes the case.

$\square$

Task 2.4 (4%). Prove or disprove Theorem 2.

Theorem 2. *For all* `l1 : int list` *and* `l2 : string list`,

$$\texttt{unzip(zip (l1,l2))} \cong \texttt{(l1,l2)}$$

Solution 2.4 This is false. Consider the case where `l1 = [1,2]` and `l2 = ["apple"]`. Then `unzip(zip(l1,l2))` evaluates as follows:

```
  unzip(zip([1,2],["apple"]))
≅ unzip(case ([1,2],["apple"]) of ...  (1::[2],"apple"::[]) => ...)
≅ unzip((1,"apple")::zip([2],[]))
≅ unzip((1,"apple")::[])
≅ ([1],["apple"])
```

`[1]` does not equal `[1,2]`. Therefore `unzip(zip(l1,l2))` does not always evaluate to the original lists `(l1,l2)`.

3 Conway's Lost Cosmological Theorem

3.1 Definition

If l is any list of integers, the look-and-say list of s is obtained by reading off adjacent groups of identical elements in s. For example, the look-and-say list of

$$l = [2, 2, 2]$$

is

$$[3, 2]$$

because l is exactly "three twos.". Similarly, the look-and-say sequence of

$$l = [1, 2, 2]$$

is

$$[1, 1, 2, 2]$$

because l is exactly "one ones, then two twos."

We will use the term *run* to mean a maximal length sublist of a list with all equal elements. For example,

$$[1, 1, 1] \qquad \text{and} \qquad [5]$$

are both runs of the list

$$[1, 1, 1, 5, 2]$$

but

$$[1, 1] \qquad \text{and} \qquad [5, 2] \qquad \text{and} \qquad [1, 2]$$

are not: $[1, 1]$ is not maximal, $[5, 2]$ has unequal elements, and $[1, 2]$ is not a sublist.

You will now define a function `look_and_say` that computes the look-and-say sequence of its argument using a helper function and a new pattern of recursion.

3.2 Implementation

To help define the `look_and_say` function, you will write a helper function `lasHelp` with the following spec. `lasHelp` takes three arguments

- `l : int list`, the tail of the list
- `x : int`, the number found in the current run
- `acc : int`, the number of times the current number has already been seen in the run.

From these arguments, the `lasHelp` computes the pair (`tail`, `total`) where

- `tail : int list` is the tail of `l` following the last number equal to `x` at the front of the list

- `total : int` is the total length of the current run (*i.e.*, the sum of `acc` and the length of the run of numbers equal to `x` at the front of `l`).

For example,

$$\texttt{lasHelp}([1,2,3],4,1) \cong ([1,2,3],1)$$
$$\texttt{lasHelp}([2,2,6,3],2,2) \cong ([6,3],4)$$

Task 3.1 (10%). Write the function

```
lasHelp : int list * int * int -> int list * int
```

according to the given specification. Note that you can use the function `inteq` in `hw03.sml` to compare integers for equality. Now, write the function

```
look_and_say : int list -> int list
```

using this helper function.[1]

Solution 3.1 See `hw03-sol.sml`

3.3 Cultural Aside

The title of this problem comes from a theorem about the sequence generated by repeated applications of the "look and say" operation. As `look_and_say` has type `int list -> int list`, the function can be applied to its own result. For example, if we start with the list of length one consisting of just the number `1`, we get the following first 6 elements of the sequence:

```
[1]
[1,1]
[2,1]
[1,2,1,1]
[1,1,1,2,2,1]
[3,1,2,2,1,1]
```

Conway's theorem states that any element of this sequence will "decay" (by repeated applications of `look_and_say`) into a "compound" made up of combinations of "primitive elements" (there are 92 of them, plus 2 infinite families) in 24 steps. If you are interested in this sequence, you may wish to consult [**?**] or other papers about the "look and say" operation.

[1] *Hint:* The recursive call in the inductive case of `look_and_say` will sometimes be on a list that is more than one element shorter. This corresponds to the notion of well-founded recursion discussed in lecture.

4 Prefix-Sum

The prefix-sum of a list `l` is a list `s` where the i^{th} element of `s` is the sum of the first $i+1$ elements of `l`. For example,

```
prefixSum [] ≅ []
prefixSum [1,2,3] ≅ [1,3,6]
prefixSum [5,3,1] ≅ [5,8,9]
```

Task 4.1 (5%).
Implement the function

```
prefixSum : int list -> int list
```

that computes the prefix-sum. You must use the `add_to_each` function provided, which adds an integer to each element of a list, and your solution must be in $O(n^2)$ but *not* in $O(n)$. This implementation will be simple, but inefficient.

Solution 4.1 See `hw03-sol.sml`

Task 4.2 (5%). Write a recurrence for the work of `prefixSum`, $W_{\texttt{prefixSum}}(n)$, where n is the length of the input list. Give a closed form for this recurrence. Argue that your closed form does indeed indicate that $W_{\texttt{prefixSum}}(n)$ is $O(n^2)$.

You may use variables $k_0, k_1, \ldots$ for constants. You should assume that `add_to_each` is a linear time function: `add_to_each l` evaluates to a value in kn steps where n is the length of `l` and k is some constant; your recurrence should involve the constant k.

Solution 4.2

$$W_{\text{PS}}(n) = \begin{cases} k_0 & \text{if } n = 0 \\ k_1 n + W_{\text{PS}}(n-1) & \text{if } s > 0 \end{cases}$$

The closed form of this recurrence is

$$W_{\text{PS}}(n) = k_2 n^2 + k_2 n + k_0$$

We have seen in lecture that this type of recurrence is in the set $O(n^2)$. Thus `prefixSum` runs in $O(n^2)$ time.

The above is a fine answer to a "what is the recurrence" question. However, if you are curious how we derived this recurrence from the code, here is an explanation: In the case for [], the code steps as follows:

```
prefixSum []
|-> case [] of [] => [] | x :: xs => x :: add_to_list (prefixSum xs, x)
|-> []
```

Therefore, $W_{\text{PS}}(0) = 2$, which is constant.

In the case for `x::xs`, we calculate as follows:

```
prefixSum (n::l)
|-> case n::l of nil => nil | x :: xs => x :: add_to_list (prefixSum xs, x)
|-> x :: add_to_list (prefixSum xs, x)
|->* x :: add_to_list (l1, n)
|->* l2
```

As it takes $W_{\text{PS}}(n-1)$ steps to evaluate `prefixSum l` and it takes $k(n-1)$ steps to evaluate `add_to_list`, it follows that $W_{\text{PS}}(n) = 2 + k(n-1) + W_{\text{PS}}(n-1)$.

It is sufficient to replace the numbers with constants, k_0 and k_1, yielding the above recurrence.

To find the closed form of the recurrence, we can visualize it by expanding the recurrence as in lecture. By doing so we get

$W_{\text{PS}}(n) = k_1 n + k_1(n-1) + k_1(n-2) + ... + k_0$

This is equivalent to $\sum_{i=0}^{n} k_1 i + k_0$. We can expand this using properties of aritmetic and sums to get

$$\begin{aligned}\sum_{i=0}^{n} k_1 i + k_0 &= k_1 \frac{n(n+1)}{2} + k_0 \\ &= \frac{k_1}{2} n^2 + \frac{k_1}{2} n + k_0 \\ &= k_2 n^2 + k_2 n + k_0\end{aligned}$$

This closed form is $O(n^2)$, as it is clearly quadratic, and the n^2 term will dominate for large n.

In order to compute the prefix sum operation in linear time, we will use the technique of adding an additional argument: *harder problems can be easier.*

Task 4.3 (10%). Write the `prefixSumHelp` function that uses an additional argument to compute the prefix sum operation in linear time. You must determine what the additional argument should be. Once you have defined `prefixSumHelp`, use it to define the function

```
prefixSumFast : int list -> int list
```

that computes the prefix sum.

Solution 4.3 See `hw03-sol.sml`

Task 4.4 (5%). Write a recurrence for the work of `prefixSumFast`, $W_{\texttt{prefixSumFast}}(n)$, where n is the length of the input list. Give a closed form for this recurrence. Argue that your closed form does indeed indicate that `prefixSumFast` is in $O(n)$.

Solution 4.4 Let $W_{\text{PSH}}(n)$ be the time for `prefixSumHelp` on a list of length n, and $W_{\text{PSF}}(n)$ be the time for `prefixSumFast` on a list of length n.

$$W_{\text{PSH}}(n) = \begin{cases} k_0 & \text{if } n = 0 \\ k_1 + W_{\text{PSH}}(n-1) & \text{if } n > 0 \end{cases}$$

$$W_{\text{PSF}}(n) = k_2 + W_{\text{PSH}}(n)$$

The closed form of this recurrence is

$$W_{\text{PSF}}(n) = k_3 + k_1 n$$

Clearly this closed form is $O(n)$, as the dominating term is a factor of n for large values of n. Thus `prefixSumFast` must run in $O(n)$ time.

5 Sublist

When programming with lists, we often need to work with a segment of a larger list. For example, one might need to access only the last three elements of a list or only the middle element. Any such segment is called a *sublist*.

More formally: if `L` is any list, we say that `S` is a sublist of `L` starting at i if and only if there exist `l1` and `l2` such that

$$\texttt{l1@S@l2} \cong \texttt{L}$$

and

$$\texttt{length l1} \cong \texttt{i}$$

For example, $[1, 2]$ is a sublist of $[1, 2, 3]$ starting at 0 because

$$[]@[1, 2]@[3] \cong [1, 2, 3] \qquad \text{and} \qquad \texttt{length } [] \cong 0$$

Task 5.1 (2%). The spec for a function that computes sublists as defined above will have the form:

> For all `l:int list`, `i:int`, `k:int`, if __________________________ then there exists an `S` such that `S` is the sublist of `l` starting at `i`, and
>
> $$\texttt{length S} \cong \texttt{k}$$
>
> , and
>
> $$\texttt{sublist(i, k, l)} \cong \texttt{S}$$

The blank is called the *preconditions*, and represents assumptions about the input. Fill in the blank to complete this spec correctly.

Solution 5.1 There are three preconditions that must hold for `sublist` to work properly.

- The index i must be non-negative, meaning `0 <= i`.
- The length of the sublist k must be non-negative, meaning `0 <= k`.
- The sublist must be contained within the list `L`. That is, `i + k <= length L`

Task 5.2 (6%). Implement a function

```
sublist : int * int * int list -> int list
```

that meets the spec you gave above.

Because the spec has the form of an implication, in the body of `sublist` you should assume that whatever preconditions you required in Task 5.1 are met: if they are not, your function can do anything you want and still meet its spec!

Note that the definition above implies that we index lists from zero, so

```
sublist (0, 3, [1,2,3,4]) ≅ [1,2,3]
```

Solution 5.2 See `hw03-sol.sml`

The spec that you completed above is good because it closely mirrors the abstract notion of a sublist, but bad because it's very stringent: any code calling sublist must ensure that the assumptions about the input hold or else it will fail. Since the exact mode of failure is not documented in the type or in the spec, this can produce behaviour that's very hard to debug.

Sometimes, the caller will be able to prove that these assumptions hold because of other specification-level information. Other times, the information available at compile-time will not be enough to ensure that these assumptions are met. In these circumstances, you can use a run-time check to bridge the gap.

Task 5.3 (5%).

Implement a function

```
sublist_check : int * int * int list -> int list
```

where `sublist_check(i,k,l)` evaluates to the sublist of `l` starting at `i` with length `k` if possible, or raises an exception explaining why it's not possible.

`sublist_check` should explicitly check the preconditions you listed in Task 5.1. If all of the conditions are met, it should call `sublist` to compute the sublist—do not reimplement `sublist`! If any one is not met, it should raise a `Fail` exception with a helpful error message describing what's wrong with the arguments.

If any call that your `sublist_check` makes to `sublist` can raise an exception or produce an incorrect result: your solution is broken! This may be because your spec in 5.1 is not strong enough or you forgot to check a precondition.

Note on testing: your tests for `sublist_check` should show that it raises the appropriate exceptions when the preconditions are violated. You should do these tests at the REPL and then copy them into a comment in your `hw03.sml` file (otherwise your file will be subject to a grade penalty for not loading cleanly). We will show you how to regression-test code that raises exceptions later in the course.

Solution 5.3 See `hw03-sol.sml`

6 Subset sum

A *multiset* is a slight generalization of a set where elements can appear more than once. A *submultiset* of a multiset M is a multiset, all of whose elements are elements of M. To avoid too many awkward sentences, we will use the term *subset* to mean *submultiset*.

It follows from the definition that if U is a sub(multi)set of M, and some element x appears in U k times, then x appears in M at least k times. If M is any finite multiset of integers, the sum of M is

$$\sum_{x \in M} x$$

With these definitions, the multiset subset sum problem is answering the following question.

> Let M be a finite multiset of integers and n a target value. Does there exists any subset U of M such that the sum of U is exactly n?

Consider the subset sum problem given by

$$M = \{1, 2, 1, -6, 10\} \qquad\qquad n = 4$$

The answer is "yes" because there exists a subset of M that sums to 4, specifically

$$U_1 = \{1, 1, 2\}$$

It's also yes because

$$U_1 = \{-6, 10\}$$

sums to 4 and is a subset of M. However,

$$U_3 = \{2, 2\}$$

is not a witness to the solution to this instance. While U_3 sums to 4 and each of its elements occurs in M, it is not a subset of M because 2 occurs only once in M but twice in U_2.

Representation You'll implement three solutions to the subset sum problem. In all three, we represent multisets of integers as SML values of type `int list`, where the integers may be negative. You should think of these lists as just an enumeration of the elements of a particular multiset. The order that the elements appear in the list is not important.

6.1 Basic solution

Task 6.1 (12%). Write the function

```
subset_sum : int list * int -> bool
```

that returns `true` if and only if the input list has a subset that sums to the target number. As a convention, the empty list `[]` has a sum of 0. Start from the following useful fact: each element of the set is in the subset, or it isn't.[2]

Solution 6.1 See `hw03-sol.sml`

6.2 NP-completeness and certificates

Subset sum is an interesting problem because it is *NP-complete*. NP-completeness has to do with the time-complexity of algorithms, and is covered in more detail in courses like 15-251, but here's the basic idea:

- A problem is in P if there is a polynomial-time algorithm for it—that is, an algorithm one whose work is in $O(n)$, or $O(n^2)$, or $O(n^{14})$, etc.
- A problem is in NP if an affirmative answer can be *verified* in polynomial time.

Subset sum is in NP. Suppose that you're presented with a multiset M, another multiset U, and an integer n. You can easily *check* that the sum of U is actually n and that U is a subset of M in polynomial time. This is exactly what the definition of NP requires.

This means we can write an implementation of subset sum which produces a *certificate* on affirmative instances of the problem—an easily-checked witness that the computed answer is correct. Negative instances of the problem—when there is no subset that sums to n—are not so easily checked.

You will now prove that `subset_sum` is in NP by implementing a certificate-generating version.

Task 6.2 (8%). Write the function

```
subset_sum_cert : int list * int -> bool * int list
```

which, if the input multiset `M` has a subset that sums to the target number `n`, returns (`true`, `U`) where `U` is a subset of `M` which sums to `n`. If no such subset exists, it should return (`false`, `nil`).[3]

Solution 6.2 See `hw03-sol.sml`

[2] *Hint:* It's easy to produce correct and unnecessarily complicated functions to compute subset sums. It's almost certain that your solution will have $O(2^n)$ work, so don't try to optimize your code too much. There is a very clean way to write this in a few (5-10ish) elegant lines.

[3] You'll note that the empty list returned when a qualifying subset does not exist is superfluous; soon, we'll cover a better way to handle these kinds of situations, called `option` types.

NP-Completeness The P = NP problem, which is one of the biggest open problems in computer science, asks whether there are polynomial-time algorithms for *all* of the problems in NP. Right now, there are problems in NP, such as subset sum, for which only exponential-time algorithms are known. However, subset sum is *NP-complete*, which means that if you could solve it in polynomial time, then you could solve all problems in NP in polynomial time, so P = NP.[4]

6.3 Double Checking

`subset_sum_cert` produces a certificate C along with its boolean response. Consequently, we can use it to build trust-worthy code, without knowing anything about the correctness of `subset_sum_cert`. This is a process called *double-checking.*

Suppose `subset_sum_cert` might be buggy. We can still use it in client code by double-checking the certificate C. For example, consider some client code that calls `subset_sum_cert`. If it's buggy, it might say `false` when there is a subset that sums to the target, or it might say `true` when there isn't. But in the latter case, it has to give you a certificate C, which you can independently check. If this certificate passes the checks, then you know that even though `subset_sum_cert` might *sometimes* give you the wrong answer, in this particular case, it did the right thing.

Task 6.3 (5%). We can wrap this double-checking up as a function

```
subset_sum_dc : int list * int -> bool
```

which is a double-checking version of subset sum. It should call `subset_sum_cert` on its input. In the affirmative instances, it should then check the certificate that was produced, and return `true` when the certificate is valid, and raise an exception `Fail "invalid certificate"` when the certificate fails to verify. For the negative instances, `subset_sum_cert` does not return a certificate, so `subset_sum_dc` should just return `false`.

For checking the certificate, we have provided functions `sum_list : int list -> int`, which sums a list, and `contained : int list * int list -> bool`, which determines whether its first argument is a submultiset of its second.

Solution 6.3 See `hw03-sol.sml`

Because only the affirmative instances are certified, the best we can say about the behavior of `subset_sum_dc` is the following *partial correctness spec:*

Theorem 3. *If* `subset_sum_dc (s,t)` $\cong$ `true` *then there is some subset of* `s` *that sums to* `t`.

This says that `subset_sum_dc` is correct (with respect to the mathematical definition of subset-sum) when it returns `true`. You can prove this without reasoning about the

[4] So: extra credit, several million dollars, and a PhD, for a polynomial time algorithm.

correctness of `subset_sum_cert`. Theorem 3 intentionally does not say anything about what happens when `subset_sum_dc` returns `false`, because we cannot establish correctness in this case without reasoning about the code of `subset_sum_cert`.

Task 6.4 (3%). Prove Theorem 3. Your proof should be very short, and should correctly prove the result even if there is a bug in your implementation of `subset_sum_cert`.

As lemmas, you may assume that `sum_list` and `contained` behave as described above, and you may assume the following fact about equivalence:

> If `case e of <branches>` is valuable, then `e` is valuable.

Be sure to cite when you use these lemmas.

Solution 6.4

For reference, here is the code:

```
fun subset_sum_dc (l : int list, s : int) : bool =
    case subset_sum_cert (l, s) of
      (true, l') => (case inteq (s, sum_list l') andalso contained (l', l) of
                        true => true
                      | false => raise Fail "invalid certificate")
    | (false, _) => false
```

Here's the idea of the proof: we assume that `subset_sum_dc(l,s) == true`. This means it must have taken the branch where `inteq (s, sum_list l') andalso contained (l'` is true: the other two branches raise an exception or return false, which contradicts the assumption that it evaluates to `true`. When `inteq (s, sum_list l')` and `contained (l', l)` are true, the specs for these functions tell us that `l'` sums to `s` and is a sublist of `l`, which is what we needed to show.

We can prove this formally as follows:

Proof. Suppose `subset_sum_dc (l,s) == true`. We know that

```
    subset_sum_dc (l,s)
==  case subset_sum_cert (l, s) of
      (true, l') => (case inteq (s, sum_list l') andalso contained (l', l) of
                        true => true
                      | false => raise Fail "invalid certificate")
    | (false, _) => false
```

We'll call this whole case expression `E`. By transitivity, we know that `E == true`.

In the problem statement, we told you to use the fact that if a `case` is equal to a value, then the expression being cased-on (the "scurtinee") is valuable (the intuition is that, if the scurtinee diverged or raised an exception, the whole `case` would, and therefore would not be equal to a value). Here, we know that `subset_sum_cert (l, s)` is equal to some value. Every value of type `bool * int list` is of the form `(v1 : bool, v2 : int list)`, and every value of type `bool` is either `true` or `false`. So `subset_sum_cert (l, s)` is either equal to `(true,l')` or `(false,l')` for some value `l' : int list`. So we have two cases:

- In the first, `subset_sum_cert (l, s) == (true,l')` for some `l'`, so

  ```
     E
  == case (true,l') of
         (true, l') => (case inteq (s, sum_list l') andalso contained (l', l) of
                            true => true
                          | false => raise Fail "invalid certificate")
       | (false, _) => false                                                    [step]
  == (case inteq (s, sum_list l') andalso contained (l', l) of
                            true => true
                          | false => raise Fail "invalid certificate")
  ```

  Call the case expression in the last step `F`. We know by assumption that `subset_sum_dc (l, s) == E == true`, so `F == true` by transitivity.

  Because `F` is valuable, the scrutinee `inteq (s, sum_list l') andalso contained (l', l)` is valuable, and because the only values of type `bool` are `true` and `false`, we have two subcases:

  - In the first,

    `inteq (s, sum_list l') andalso contained (l', l) == true`

    By definition of `andalso`, this means that `inteq (s, sum_list l') == true` and `contained (l', l) == true`. By the lemmas given in the problem, the former implies that the sum of `l'` is `s`, and the latter implies that `l'` is a submultiset of `l`. Thus, `l'` is a subset that sums to `s`, as we needed to show.
  - In the second,

    `inteq (s, sum_list l') andalso contained (l', l) == false`

    so

    ```
       E
    == (case false of
            true => true
          | false => raise Fail "invalid certificate")
    == raise Fail "invalid certificate"                    [step]
    ```

    But we know that `subset_sum_dc(l,s) == E == true` by assumption, so by transitivity `raise Fail "invalid certificate" == true`. This is a contradiction, because one side raises an exception and the other is a value. Because anything follows from a contradiction, the result holds in this case.

- In the second, `subset_sum_cert (l, s) == (false,l')` for some `l'`. Then

```
   E
== case (false,l') of
      (true, l') => (case inteq (s, sum_list l') andalso contained (l', l) of
                        true => true
                      | false => raise Fail "invalid certificate")
    | (false, _) => false                                              [step]
== false
```

  But we know by assumption that `subset_sum_dc(l,s) == E == true`, so this implies that `true == false`, which is a contradiction. Because anything follows from a contradiction, the result holds in this case.

□