FPP Term Project Report

Yale-NUS College AY 17/18

Jeremy Yew
A0156262H

6 December 2017

1.Table of Contents

2. 2 x 2 matrices

- **Property 10 (Associativity of multiplication):**
  - Expansion and then arithmetic manipulation. The lemma shuffle_helper stated precisely the shuffle pattern I required. Even though I still needed to give it arguments each time, it is still shorter than using the less precise shuffle lemmas provided by Coq.
- **Property 12 (I is neutral on the left and right of multiplication):**
  - The proof steps are almost identical in both except that we rewrite multiplications of 1 and 0 on the right instead of the left in **I_neutral_mul22_r**. This simply reflects the fact that the 0's in I are on the right in the multiplication.
- **Proposition 29 (M x M^n is commutative):**
  - This is why the proof about the exponentiation of M 1 1 0 1 works with both definitions of exponentiation.
  - **Exercise 31** also works because both definitions of exponentiation are equivalent.
- **Proposition 38 (transposition and exponentiation of M is commutative):**
  - This is true because transposition is distributive over multiplication; on the RHS, transposing an exponentiation involves transposing both M and M^n, which flips the order of the arguments to be multiplied. After which Proposition 29 helps us switch it back to match LHS.
- **Exercise 40 (Proof of Proposition 33 using Proposition 14):**
  - We may use Proposition 14 by doubly-transposing the matrices in LHS and RHS (the equation is the same since transposition is involutive). The singly-transposed arguments then reflect the statement of Proposition 14. After which Proposition 38 helps switch the order of operations for RHS to match LHS.
- **Exercise 25 (Powers of F):**
  - Since F^n effectively returns three consecutive Fibonnaci numbers, we might use F^n to implement another function fib_v4 which calculates Fibonacci numbers iteratively. It is similar to fib_v3 (from week 4), in that F^n only makes n recursive calls (linear time), as opposed to much more with fib.
  - We may also implement fib_v5 which is also iterative, but makes 2 less calls than fib_v4, capitalizing on the fact that F^n stores both the n^{th} and $n+1^{th}$ Fibonacci numbers, which can be added to compute the $n+2^{th}$ Fibonacci number.
- **Subsidiary Question (fibonacci_addition_of_powers_of_F):**
  - The proposition stated is simply due to the fact that the elements of the matrix are successive Fibonacci numbers – the same recurrence relation is thus reflected in the powers of F.
- **In general:**
  - Unfold lemmas are only for recursive functions.

- With arithmetic expansion in the context of this exercise, intuitively it seems to me the most efficient to rewrite in the following order:
  - distributivity and associativity of multiplication,
  - associativity of addition,
  - multiplications of 0 on the right or left,
  - multiplications of 1 on the right or left,
  - additions of 0 on the right or left and/or all additions of 1 on the right or left.

3. A commutative diagram

- Task 1 (evaluate):
  - **At most one:** We prove by inducting on arithmetic expressions. For Plus ae1 ae2 and Minus ae1 ae2, we then prove for each of the cases where (f ae1) and (f ae2) return Expressible_nat n1 and Expressible_msg s1 respectively. We must also apply our case assumptions about (f ae1) and (f ae2) after rewriting with the specifications because each specification depends on those assumptions.
  - **At least one:** We successively split the nested conjunctions of specifications, and use the assumptions to rewrite the unfolding of evaluate.
- Task 1 (interpret):
  - **At most one:** We specify that there is only one case of source program, Source_program ae (by definition). We then rewrite using the respective specifications of interpret for f and g, supplying this assumption with evaluate as well as the proven theorem that evaluate satisfies its specification.
  - **At least one:** We prove that given some hypothetical implementation of evaluate that satisfies its own specification, our implementation of interpret, which uses our implementation Top.evaluate, produces the same output as the hypothetical evaluate. We use our theorems about evaluate to prove that our implementation Top.evaluate is equivalent to the hypothetical evaluate, since both implementations indeed satisfy the specification of evaluate.
- Task 3 (decode_execute):
  - **At most one:** We prove by inducting on byte code instructions. For ADD and SUB, we then prove for the cases where ds is nil, (n2 :: ds'), or (n2 :: n1 :: ds'') respectively. We do not need to apply our assumptions as the specifications do not depend on separate premises. For SUB, when ds = (n2 :: n1 :: ds''), we also prove for the cases when (ltb n1 n2 is true or false respectively. Here we must apply our assumptions because the specifications do depend on a separate premise about (ltb n1 n2).

- **At least one:** We successively split the nested conjunctions of specifications, and show that the unfolded decode_execute produces the specified output. Again, we only need to rewrite using the assumptions about (ltb n1 n2) for SUB when ds = (n2 :: n1 :: ds'') because that is the only case where the specification depends on a separate premise.

- **Task 4 (fetch_decode_execute_loop):**
  - **At most one:** We have to destruct the specifications supplied with our implemented decode_execute and the theorem that it satisfies its own specification, instead of immediately introducing them as conjunctions, since the specification for decode_execute contains the separate premise about decode_execute. We then prove via inducting on byte code instructions, and then for the cases where (decode_execute bci ds) returns a OK ds' or a KO s.
  - **At least one:** We successively split the nested conjunctions of specifications, and rewrite using our unfold lemmas for fetch_decode_execute_loop. For the inductive cases of bcis, we first unfold fetch_decode_execute_loop, and then prove that our implementation Top.decode_execute used by our implementation of fetch_decode_execute_loop is equivalent to the hypothetical decode_execute, since both implementations indeed satisfy their specifications. We can then apply the assumptions about the hypothetical decode_execute to the unfolded fetch_decode_execute_loop in order to obtain the specified output.
- **Task 5 (fetch_decode_execute_loop_is_distributive_over_append):**
  - Incomplete. Not sure if to use an existential or not in the statement of the theorem, and not sure how to use the inductive hypothesis to prove the inductive case.
- **Task 6 (run):**
  - **At most one:** Incomplete. We have to destruct the specifications supplied with our implemented fetch_decode_execute_loop and the theorem that it satisfies its own specification, instead of immediately introducing them as conjunctions, since the specification for run contains the separate premise about fetch_decode_execute_loop. We then specify that tp has only one case, Target_program bcis, by definition. We then prove for the cases where (decode_execute bci ds) returns a OK ds' or a KO s. I got stuck here because it seems to instantiate Datatypes.nil instead of nil.
  - **At least one:** We successively split the nested conjunctions of specifications. We first unfold run, and then prove that our implementation Top.fetch_decode_execute_loop used by our implementation of run is equivalent to the hypothetical fetch_decode_execute_loop, since both implementations indeed satisfy their specifications. We can then apply the assumptions about the

hypothetical fetch_decode_execute_loop to the unfolded run in order to obtain the specified output.

- **Task 6 (compile_aux):**
    - **At most one:** We prove by inducting on arithmetic expressions.
    - **At least one:** We successively split the nested conjunctions of specifications, and rewrite with the unfold lemmas for compile_aux.
- **Task 7 (compile):**
    - **At most one:** We specify that sp has only one case, Source_program ae, by definition. We then rewrite using the specifications, supplied with the theorem that our implemented compile_aux satisfies its own specification.
    - **At least one:** We prove that our implementation Top.compile_aux used by our implementation of compile is equivalent to the hypothetical compile_aux, since they both indeed satisfy their specifications.
- **Task 9**
    - Incomplete.

4. Acknowledgements

Thank you Prof Danvy for your kind patience and generosity. It has been a fruitful experience.