# A note about teaching CS3234 this semester

Olivier Danvy <olivier@comp.nus.edu.sg>

February 24, 2025

**Abstract**

This note illustrates how the six types of learning objectives in Bloom et al.'s hierarchical taxonomy of student-learning objectives are reached in the instance of CS3234 in Sem2 of AY2024/25. The illustration is based on quotes from weekly handins.

## 1 Bloom et al.'s hierarchical taxonomy of student-learning objectives

The present rendering of CS3234 ambitiously aims to reach all of the six types of learning objectives in Bloom et al.'s hierarchical taxonomy, namely knowledge, comprehension, application, analysis, synthesis, and evaluation.

**Knowledge:** The material is organized like a snowball and so the students continuously recall the ideas they have been exposed to so far because they keep reusing it to investigate the new ideas of each new week.

Here is an excerpt of a report for Week 02 this semester:

> The aim of this exercise was to deepen our understanding of polymorphic types and the effective handling of optional values. Additionally, it reinforced practices for structuring functions and writing high-coverage unit tests to ensure reliability and scalability.
>
> Through this exercise, we observed that many of the implemented functions were built by composing core polymorphic comparison functions. This approach extended to unit tests as well. By leveraging anonymous functions, such as `fun t0 ... => ...`, we were able to extract and reuse existing comparison logic, applying previously tested functions rather than redefining redundant test cases.
>
> This highlights the importance of creating modular function/components to ensure scalability, clarity and correctness. This also emphasize the importance of polymorphism as it would be much harder building such component without it.

Here is another excerpt of a report for Week 02 this semester:

> Let us summarize what we have learnt. First, we learnt about tail-recursive functions, and how we make functions tail-recursive using accumulators and continuations. Then, we learnt about polymorphism in Gallina and how we can write polymorphic structures and functions. Lastly, we learnt about writing formal statements in the Coq Proof Assistant, and stated formal claims about the soundness and completeness of some functions.

Here is a third excerpt of a report for Week 02 this semester:

> It may be alarming that the second report has been submitted, but we have not seen a single proof. Yet, this shows that our initial understanding of proofs may have been truncated and that these exercises are meant to build the foundation for us to understand what "proofs" really are.

Here is a fourth excerpt of a report for Week 05 this semester:

> The writing of implementation and tests in this exercise nicely ties up several concepts we have learnt so far, helping us identify any knowledge gaps or misconceptions in preparation for the mini-project.
>
> The proofs of soundness and completeness provided rich opportunities for practice with fold-unfold lemmas, reasoning by cases and induction, knowing to clear suitable premises, finding useful library lemmas, and use of the discriminate and injection tactics.

**Comprehension:** The students continuously demonstrate their understanding of the ideas they are exposed to by using them and explaining their uses in their weekly handins.

Here is an excerpt of a report for Week 02 this semester:

> Exercise 00: Reflecting on the contents of the past week
>
> This week, we explore several key concepts in functional programming and formal verification. We started by understanding how accumulators can be used to build out results iteratively, and how quantifiers, specifically the existential quantifier (`exists`) and the universal quantifier (`forall`), allow us to express properties about the existence or universality of values. We also learned about implicit parameters like those in `Some` and `None` which simplify definitions, and the power of polymorphism which helps to encompass polymorphic data types and polymorphic functions within Gallina. Lastly, we used the tCPA command `Search` to explore and prove other properties within our formal developments.

Here is an intermediate conclusion in a report for Week 05 this semester:

> This exercise gave us good practice with fold-unfold lemmas as well as some of the newer tactics in our toolbox, such as `injection` and `discriminate`. More importantly, we have also learnt that the formulation of a Eureka lemma is very important, as a poorly-formulated lemma may not give us enough information to prove meaningful properties about our programs.

Here is another intermediate conclusion in a report for Week 05 this semester:

> In completing this exercise, we have made use of many new tactics introduced in this week's lecture, including fold-unfold lemmas, `injection`, and `discriminate`, in order to design and prove properties about the mirror predicate. This exercise provides practice in using these new tools, as well as deepening our understanding of the many properties that a polymorphic binary tree has. As with all other exercises, this exercise also builds muscle memory and frequently brings back ideas and concepts introduced previously. While the food for thought in Section 3.11 was easy to conceptualize, it was difficult to actualize in tCPA. Nevertheless, we are prompted to think further about how to represent our thoughts in formal logical statements to prove them effectively.

**Application:** The students permanently put the ideas they are exposed to into programming and proving practice.

Here is an excerpt of a report for Week 02 this semester:

> The `mult_acc` function demonstrates how recursion can be combined with an accumulator to compute multiplication in an iterative manner. The approach above uses an accumulator to store intermediate results and builds towards the final answer step by step. The `mult_alt` acts as a convenient wrapper function around `mult_acc` by setting the initial value of the accumulator. The correctness of our implementation is also checked against the `test_mult` function which verifies key properties of multiplication, including commutativity, the zero property, and its relationship to repeated addition. Overall, this example illustrates how accumulators can be used in recursive definitions effectively.

And here is an intermediate conclusion in a report for Week 05 this semester:

> This series of exercises on writing routine inductions on both recursive definitions and accumulator-based ones have proven to be most enlightening. We found many interesting discoveries and deepened our understanding in unexpected manners, e.g., writing forward vs. backward proofs. We also have grasped the pros and cons of accumulators. While accumulators may make implementations faster/more efficient, they come at a cost of being harder to reason about, requiring more careful thought. In fact, most of

> the accumulator-based proofs ended up relying on proving recursive-based lemmas to aid in the proof. It has also been useful to understand the axiomatic principles behind how addition and multiplication is defined, and how these definitions are then used to prove properties that is taken for granted.

**Analysis:** In each report, the students are expected to contrast and relate the new ideas they are exposed to with the ideas they were exposed to previously, be it earlier in the semester or in their previous courses.

Here is an excerpt of a report for Week 02 this semester:

> This exercise highlights that it is sometimes not possible to simply transform a recursive function into a tail-recursive version using an accumulator. One requirement for the correct implementation of such an accumulator function is that the operation performed on the result of the recursive call must be both commutative and associative. Given that subtraction is not associative and not commutative, it is a good negative example to showcase this point: if the series of operations are reordered, the final computed result will (in general) be different. As such, it is important to consider the properties of an operation when designing recursive and accumulator-based functions.

Here is another excerpt of a report for Week 02 this semester:

> This is where our knowledge of CS1231(S) and/or MA1100(T) can finally be formalized in the Coq Proof Assistant. This section familiarises us with the top-level forms of several logical statements, including: theorems, lemmas, propositions, corollaries and properties. Aside from this, it also allows us to check logical operations such as conjunctions (AND), disjunctions (OR), negations (NOT), implications and bi-implications.

Here is a third excerpt of a report for Week 05 this semester:

> This exercise summarises various new tactics and proof patterns. In retrospect, particular attention is paid to building new knowledge on top of existing knowledge, and understanding meaning in spite of the mechanized nature of what we are doing.

**Synthesis:** The students permanently combine the ideas they are exposed to. For example, they have never stated theorems and lemmas before, and they appreciate to use their testing skills to check whether each statement is not wrong prior to embarking on a proof.

Here is an excerpt of a report for Week 02 this semester:

> This exercise helped us gain more experience expressing informal conjectures as formal theorems in Coq, especially with stating theorems.

Here is another excerpt of a report for Week 02 this semester:

> This exercise provides a formal specification for the Boolean negation function along with two distinct implementations. The soundness and completeness theorems establish the correctness of the implementations with respect to both the specification and the built-in `negb` function, while the equivalence theorem demonstrates the functional equality of our two implementations. Our observation about the symmetry of soundness and completeness highlights a key property of equality and its implications for reasoning about program correctness. While the proofs are admitted, the theorems themselves provide a rigorous foundation for reasoning about Boolean negation in Coq.

And here is the conclusion of a report for Week 05 this semester:

> This week, in addition to expanding our repertoire of tCPA tactics, we dedicated significant time to writing proofs, including those involving more complex structures such as polymorphic binary trees and comparative analyses of different definitions of the addition and multiplication functions for Peano numbers. The key concepts emphasized this week include:
>
> Structured Proof Organisation: We were encouraged to actively formulate auxiliary lemmas, or "Eureka" lemmas, when approaching non-trivial proofs and to maximize the reuse of existing lemmas whenever possible to streamline proof construction.
>
> Computation: In certain cases, achieving term normalization in proof goals requires a sequence of verbose reduction steps. The Coq Proof Assistant provides the specialized `compute` tactic to automate this process. However, we must ensure that no additional non-computational steps are needed afterwards to avoid misusing.
>
> Directionality of Proofs: Proof construction follows two paradigms – forward and backward reasoning – each supported by distinct sets of tactics. Forward proofs typically utilize tactics such as `assert` and `destruct`, while backward proofs leverage `apply`. The `rewrite` tactic can be applied in both directions. Choosing the most suitable approach enhances the intuitive coherence of the proof.
>
> Fold-Unfold Lemmas: In instructional contexts, rather than using the `unfold` tactic directly, we can apply specific fold-unfold lemmas, each corresponding to a distinct case branch of a function. This approach improves clarity and facilitates a deeper understanding of proof structure and reasoning. Through these focused explorations, we have reinforced our ability to

construct structured, efficient, and comprehensible proofs within the Coq framework.

**Evaluation:** The students ceaselessly assess what they learn in CS3234 in the light of what they have learned before, and dually they ceaselessly re-assess what they have learned before CS3234 in the light of what they learn in CS3234.

Here is an excerpt of a report for Week 02 this semester:

> This exercise aims to practice the skill of using the scientific method to find out what a function does. The standard process is firstly applying a few inputs to the function and observing the output pattern. Then we can set up a hypothesis about the function and test the hypothesis. If the hypothesis passes the test, then we can formalize the proof through the Coq Proof Assistant.

Here is a second excerpt of a report for Week 05 this semester:

> Through this series of exercises, we have systematically proven various properties of addition and multiplication using different implementations in Coq. The exercises allowed us to explore the interplay between recursion, accumulator-based definitions, and fold-unfold lemmas.
>
> The fold-unfold lemmas were essential for reasoning about function definitions in a stepwise manner. By explicitly unfolding recursive calls, we could express functions in their base and recursive cases – especially for the accumulator-based functions. This made it easier to manipulate expressions with equational rewrites.
>
> Aside from fully utilising the fold-unfold lemmas, we also became familiar with writing eureka lemmas, which prevented the proofs for our primary goals from becoming difficult to read (sort of providing a level of abstraction). This was evident in the proofs for commutativity and distributivity, which required auxiliary lemmas that were not immediately obvious from the main goal. These eureka lemmas helped us handle intricate recursive structures by breaking the proof into more manageable subgoals. For instance, proving that multiplication distributes over addition required an intermediate result that linked different recursive calls.
>
> We have explored the pros of accumulator functions in terms of efficiency in execution time in earlier weeks. Over these exercises, we saw both [Dr.] Jekyll and [Mr.] Hyde: while they were useful in allowing us to see how the intermediate value changes through each step, they often made approaching the primary goal itself less intuitive. As a result, properties that were proved using accumulator functions required a higher number of eureka lemmas, as compared to recursive ones (as we saw in Ex 31 and 39).

> By carefully structuring our proofs and leveraging both fundamental induction techniques and additional helper lemmas, we were able to establish important algebraic properties of our arithmetic functions, thus allowing us to fully 'accumulate' (forgive the pun), our knowledge of formalizing proofs using the Coq Proof Assistant over the past few weeks.

In addition, the students grow a healthy vigilance because the exercises, the midterm project, and the term project are replete with Easter eggs whose explicit goal is to elicit Aha! moments. Throughout the semester, there is no week without eye openers.

# 2 Conclusion

- If one of your reports is quoted in this note, congratulations!
- Otherwise, the quotes indicate the kind of thinking you should document in your reports to convey your knowledge about your knowledge.

Happy writing!