Electronics and Computer Science
Faculty of Physical Sciences and Engineering
University of Southampton

COMP6255 - Coursework 1

Kristian Ivanov - kii1u20@soton.ac.uk

19/03/2024

Dependently-typed Programming

Dependently-typed programming languages allow types in a programme to be dependent on values, allowing the type system to express and enforce more complex programme features. For example, one could construct a type for "a list of integers with exactly n elements" where n is a number. This enables extremely precise types, which can reduce errors at compile time while ensuring that the programme operates as intended at runtime.

In comparison, refinement types are types that have been augmented with predicates that limit the range of potential values. They are viewed as a subset of dependent types in which the logic is limited to decidable fragments, allowing for automatic verification without the need for manual proof. Refinement types can be easier to utilise and integrate into existing codebases by utilising SMT solvers for automated proof checking.

Pros of Using Dependent Types Over Refinement Types:

- **Expressiveness:** Dependent types can express more complex properties and invariants within the type system.
- **Proof Assistance:** They often come with proof assistants, which help in constructing proofs for program correctness.

Cons of Using Dependent Types Over Refinement Types:

- **Complexity:** The increased expressiveness can lead to more complex code and a steeper learning curve.
- **Manual Proof Writing:** More sophisticated type checking may require more manual intervention to write and verify proofs.

In essence, dependently-typed programming provides a powerful mechanism for verifying programme correctness, but it is also difficult and demanding. Refinement types provide a more automated and user-friendly method, but their expressiveness is limited when compared to full dependent types.

In this assignment, I used both dependent and refinement types. For example, in the F* proof of Insertion Sort, I constructed **sorted_l** (which refines the type list nat with the predicate that the list satisfies the definition of sorted), and in Coq, I defined **sorted** as a dependent type, as the value is determined by the content of the list supplied.

To conclude, refinement types are more easily understood and defined. While dependent types provide greater expressive ability, they are not always required and can add complexity if used unnecessarily.

Insertion Sort

To prove the correctness of insertion sort in Coq, I needed to prove that the algorithm produces a permutation of the input list, and the result it produces is sorted. Following the tutorial from Software Foundations proved easy to do, with the most crucial tactics being **bdestruct** and **lia**. These two tactics are available out-of-the-box, and make the proof a lot easier.

To translate the Coq proof to F*, I began by translating the definitions of sorted and **is_a_sorting_algorithm**. The translation is extremely similar to the syntax used in the Coq proof. I translated every Coq function and lemma, and I created one new lemma, **cons_insert_perm**, which is discussed in the code. I defined one global refinement type, **type sorted_list = l:list nat {sorted l}**, as well as others throughout the proof. Both proofs use induction, but the F* code is shorter and easier to grasp. This is not always the case, as F* does not provide information about what it is doing behind the scenes, making following even somewhat lengthier proofs difficult. I found myself grabbing pen and paper and outlining the logic of a proof, and then trying to write that using the F* syntax.

QuickSort

This algorithm proved much more difficult to translate from F* to Coq. I attempted translating the F* proof "word-for-word" at first. I determined which lemmas and functions needed to be translated: **sort**, **partition**, **mem**, **partition_mem**, **sorted_concat**, and **append_mem**. I successfully translated **sort**, **partition**, **mem**, **sorted_concat** (albeit this was quite difficult), and **append_mem**. To make the definition of **sort** work, I had to add an additional variable because Coq couldn't verify that the recursion terminated when I tried recursing simply on the input list.
The final definition looks like this:

Fixpoint quicksort (l: list nat) (n: nat){struct n} : list nat :=

The additional variable, **n**, represents the length of the input list **l**, and is used in the recursion in addition to the list. I created a second function which receives only the list as input, and calls sort with that list and its length.

Definition sort (l: list nat) : list nat := quicksort l (length l).

More detailed explanation of the **quicksort** function is available in the code itself, but it will always correctly terminate given the length of the original list. It will arrive at a case where the input list is empty, or if the partitioned lists have exactly a length of **n**, it will terminate when **n** reaches 0.
No matter what strategies I tried, I was unable to prove **partition_mem** in Coq. After translating the required functions and lemmas, I realised that we actually need to prove the algorithm using permutations, and while the **mem** function can be used to prove permutations, it is considerably tougher. So, using the definition of **is_a_sorting_algorithm** from before, I tried proving **Permutation l (sort l)**. I tried induction on the list, unfolding the definitions, applying every single tactic on permutations from the Coq standard library, and I even defined and proved numerous additional lemmas on the partition function. But, unlike with the proof of insertion sort where **simpl** or **unfold** resulted in getting **insert h (sort t)** (from the definition of sort), doing any tactic on the definition of quicksort did not break it down to the partition function. I spent well over 100 hours trying to modify the proof using every tactic defined in the Coq library, and every time I arrive as a contradiction of the type: **Permutation (hd::tl) []**, and in order for this to be true, **tl** needs to be an empty list, and the **hd** part should not exist, since in Coq **hd::[]** indicates a list with at least one element. The statement is obviously false. I searched on Google, and found an implementation of quicksort in Coq (https://gist.github.com/RyanGlScott/ff36cd6f6479b33becca83379a36ce49). It appears that the issue might be coming from the way I have defined the quicksort function. I defined the recursion on a variable n of type nat, which represents the length of the list, but the linked proof uses recursion on a complex custom dependent type. This proof is incredibly complex, with numerous lemmas, and it far from the task of translating the F* proof as closely as possible, because at that point the proofs and definitions have nothing in common.

The hardest Lemma I was able to prove was sorted_concat. At first I tried induction, destruct, and other tactics, but the statement that needed to be proves was obviously true. So I utilised the inversion tactic a lot, in order to break down the hypothesis in the context further, and was able to prove it. The difficulty comes from the way the Lemma is defined, following closely the definition in F*. It's easy for F* to compute the recursion required for the proof, but since Coq does not have an SMT solver, manual proof is required.

With that, I think an exact translation is impossible, and techniques and syntax specific to Coq must be used.