THOUGHTS

1. Software foundations

Chapters I got to read: Preface, Basics: Functional programming in Coq, Induction: Proof by induction, Lists: Working with structured data (not all of it).

- Syntax is very similar to OCaml structurally; some things have different names, but not too many of them.
- Proofs are guided by *tactics*. These are like hints you drop to the prover, such as 'rewrite this as the other side of this equality we already proved', 'rewrite both sides in a canonical way', etc.

2. Certification of sorting algorithms in the system coq[FM99]

- This paper verifies IMPERATIVE implementations of insertion sort, merge sort and heap sort. They say coq supports that.
- The language used is very similar to (a subset of) OCaml: we have functions, conditionals, let expressions references, and while loops.
- In-place sorting only
- Only integer values
- The general idea to prove correctness of a function is (as usual) to use *pre-conditions* – which tell us what the input is guaranteed to satisfy, and *post-conditions* – which tell us what the output should satisfy. Clearly, this doesn't require much imagination. But to deal with loops, we need a trick called a loop invariant, which, if true at each iteration, guarantees correctness. The magic is that then we don't have to unfold the loop when we do the proof, but only make sure that the code that is repeated preserves the invariant (or that it preserves a certain invariant depending on which case we're in, for some *constant* number of cases; this is called a *case* invariant). This seems to be the meat of the business in these proofs of correctness.
- the `(sorted t)` predicate tells us whether the part of list t between indices i and j is sorted.
- the `(permutation t t')` predicate tells us whether lists t and t' are permutations of each other.
  - This is done by generating the group of all permutations via transpositions; we define `(exchange t t' i j)` to be the predicate that t and t' differ only in the order of the i-th and j-th entry.
- Insertion sort:
  - Subroutine `(insertion n t)` inserts the n-th element of an array t into an already sorted sub-array. Some care is needed in defining the invariant...
  - Once we have the correctness of the above, the correctness of insertion sort follows easily...
- Quick sort:
  - Subroutine `(partition t i j)` takes the sub-part of the array t between indices i and j and rearranges it into two halves, such that all elements on the left are less than or equal to all elements on the right, and returns the index of the partitioning element (the pivot).

    - Subsubroutine `(swap i j)` just swaps the i-th and j-th guys. Easy. This is useful for `partition`.
  - Subroutine `(divide-and-conquer t)` calls `partition` on the list t and then recursively calls itself on the two halves to sort them. This is quicksort.
  - These guys didn't prove it in the case when we pick the pivot in a smart way (they always pick the first element). Possible room for us to do something new? They say it's not a huge change, just more annoying. Ugh...
- Heap sort:
  - Data structure: this represents a binary tree implicitly as an array, where the children of the i-th guy are the $2i+1$-th and $2i+2$-th guys.
  - The algorithm proceeds by making the array into a heap in-place, and then maintaining the left half as a heap and the right half as the already sorted elements, by repeatedly putting the top guy in the heap at the end of the array.
  - Predicate `(heap t n k)` checks if in a list t, the tree rooted at the k-th guy of elements of index at most n is a heap
  - Predicate `(inftree t n v k)` checks for a tree represented as above whether all elements are $\leq v$.
  - Subroutine `(downheap t k n)` takes an array t where the tree of elements rooted at the k-th element and of indices $\leq n$ is a heap, *except* possibly for the root node. It then makes a bunch of swaps that make this tree into a heap (invariants need figuring out of course).
  - Then, heapsort is easy: first build the heap using `downheap` a bunch of times, and then swap the 1st element with the N-th, $N-1$-th, etc. and rebuild the heap to the appropriate index between swaps.

3. Ideas

- It seems that coming up with the right interface for proving correctness of a sorting algorithm requries some care and thought about how the proof might go. So we can sort of guess what it could look like for timsort, but I'm still not absolutely sure.

  Here is a description of the main ideas behind timsort:
  - First we pass over the list and make sure each run is of at least some minimum length c.
  - Then we pass over it again and push the base address (that is, the index of the first element) and length of every run (this could be done in the above pass, but let's separate them for clarity). But as we push runs on the stack, we also sometimes merge consecutive runs until some invariant (that 'attempts to keep the run lengths as close to each other as possible to balance the merges' as the wikipedia page says) is satisfied. The condition is that if X, Y, Z are the lengths of the top three runs on the stack, we must have

    $$X > Y + Z \text{ and } Y > Z$$

    Here's where things get weird and the bug happens: these rules (from the wikipedia page) are actually buggy, and the guys who found the bug have some other rules.
  - So at any point we end up with a bunch of runs whose sizes grow faster than the Fibonacci sequence, i.e. at least exponentially fast; so it's easy to see there are at most logarithmically many runs at each point in time, which seems to be important for memory reasons. It's also important for running-time reasons it seems - it's much faster to merge these exponentially-increasing guys than to naively merge a list split into equal parts!

  - Okay, the wikipedia article is kind of poorly written. This `http://svn.python.org/projects/python/trunk/Objects/listsort.txt` seems better.
  - There are some memory optimizations ('galloping'), but let's ignore them for now...

  Anyway,
  - Subroutine `insertion-sort` - this is just as above.
  - Predicate `(run t i j)` - this checks if the subarray between the i-th and j-th index of t is a run, i.e. is already in sorted order.
  - Subroutine `merge` - this is supposed to take the next run and do any merges necessary to preserve the invariant...
  - Now piecing together all the parts should be easy. In the end, we end up with a sequence of runs that we merge in the obvious way.
- We could probably easily verify the correctness of algorithms that use sorting algorithms as a substantive building block. The 3-SUM problem is an example:

  Problem 1. Given an array of n integers, are there three that sum to zero?

  But that seems kind of lame...
- Modularization: it's easy to imagine how we could have a common coq module containing the language constructs the paper uses (that can probably be implemented in coq? or maybe we can just import them from somewhere?), another common module dealing with abstract properties related to sorting, like the predicates for being sorted and being a permutation, and then specific modules for the different sorts. It'd be cool if we notice similarities between the routines that we can share between modules

References

[FM99] Jean-Christophe Filliâtre and Nicolas Magaud. Certification of sorting algorithms in the system coq. *Theorem Proving in Higher Order Logics: Emerging Trends*, 1999.