Hints and tips (1)

- Break down complex preconditions, postassertions and invariants into simpler ones
- Keep specifications as simple as possible. In particular, keep the abstract data as simple as possible (simple abstract data makes for simple specifications). Where the abstract data is or contains a collection, use the built-in collection types (set, bag, seq and map) wherever possible.
- Declare classes final unless you expect to derive other classes from them - this makes it easier to reason about them.
- Use ref types sparingly. Only use them where you genuinely need to have two variables referring to the same object, such that any change made through one variable will be seen by the other.

Hints and tips (2)

- Avoid redeclaring member variables as interface selectors.
  - Try not to redeclare them as confined selectors either.
  - Once you redeclare data variables as selectors, you lose encapsulation, and you can't refine them to more efficient implementations.
- *Perfect* does not support global variables, so you may find yourself passing parameters that would otherwise be global data.
  - Create a class to contain all the unchanging variables that you may wish to use in different places
  - Then pass an instance of this class as a single parameter
  - You can easily add new variables to this class in the future
- Modularize large projects
  - Lets you verify each module independently
  - Organize your work to run verification during a meal break or overnight.

Dealing with verification failures

- You write your specification and ask PD to verify it
- PD proves most of the VCs, but flags a few with one of the following:
  - Given 'false', so proof is trivial
  - Refuted
  - Unprovable
  - Too hard

Dealing with verification failures

- **Given 'false', so proof is trivial** indicates unreachable code
  - Not all unreachable code is detected in this way
- If you are using inheritance:
  - An inherited function, selector, operator or schema may be unreachable in the context of a derived class…
  - …because the precondition calls a dynamically bound function, such that it cannot be satisfied for that class
  - Redeclare the function/selector/operator/schema as **absurd**
  - If you are satisfied that just part of the function is unreachable in that class, but not in another class, then ignore the warning

Dealing with verification failures

- **Refuted** indicates that the VC is false for all inputs
- Rarely happens
- Always indicates an error in your specification
  - Except that if you are using reals in your specification, PD 3.x and 4.0 may report spurious refutations due to rounding errors

Dealing with verification failures

- **Unprovable** indicates that the condition cannot be proven using the logic and axioms of PD
- If your specification relates to software:
  - There is almost certainly an error in your specification
  - Check the Unproven output for suggestions
- If your specification relates to more abstract system-level properties
  - This type of failure may indicate a deficiency in PD
  - If you are sure it should be provable, raise a support request

Dealing with verification failures

- **Too hard** means the prover reached its time or memory limit
- Inspect the proof failure
  - It may tell you something about what is wrong
- Construct an outline argument
  - Why are you sure it is provable?
- Rework the specification to make the verification simpler
  - Break it down into several simpler verification conditions
  - Consider adding proof hints
- Try increasing the time and/or memory limit
  - Increasing time limit beyond 1000 sec is rarely worthwhile

Inspecting the proof failure

- Always run verification with Unproven output enabled
  - Enabling *Proof* output slows down verification, but enabling *Unproven* output does not
- Right-click on the error, select **Go to proof information**
- There may be a suggestion
  - These may be spot-on, worthless, or in between
  - Don't act on them unless you understand them
- There may be an "**in the case that…**" clause
  - It means the prover did a case-split and this is the first case that failed, not that this is the only broken case

Constructing an outline argument

- How would you persuade a fellow developer that the verification condition represents a universal truth?
- Faced with a failed VC within some function or schema, the **only** things that you (or the prover) can assume when it is called are:
  - The class variables satisfy their type constraints
  - The class invariant is satisfied
  - The parameters to the method satisfy their type constraints
  - The precondition is satisfied
  - If the condition occurs in a list of preconditions, invariants or assertions, the **earlier** elements in the list are satisfied

Making the verification easier

- In preconditions, assertions, class invariants and loop invariants, declare multiple terms in preference to a conjunction
- The following are logically equivalent:

```
pre #s ~= 0, s.head ~= `a`
pre #s ~= 0 & s.head ~= `a`
```

- **But** there is one important difference:
  - Each comma-separated terms gives rise to a separate VC
- Several simple VCs are better than one complex VC
  - The individual VCs are easier to prove
  - If PD fails to prove one of them, the problem term is pinpointed

Explicit case splitting

- If the location of a proof failure falls immediately after a conditional, try moving the VC into each branch
- Example: prover reports: "Unable to prove: Specification satisfied at end of implementation" for a schema:

```
schema !foo                    schema !foo
  post …                         post …
  via                            via
    …                              …
    if [cond]: … ;                 if [cond]: … ; done;
      []:    … ;                     []:    … ; done;
    fi                             fi
  end;                           end;
```

- See web for other situations where you can case-split

Adding intermediate assertions

- If a postcondition involves sequential steps, your outline argument may involve hypotheses about intermediate states
- Make these hypotheses explicit by stating them as assertions

```
schema !foo                    schema !foo
  post A                         post A
    then B                         then (assert P; B)
    then C                         then (assert Q; C)
```

Dealing with verification failures

- Declare classes **final** except where you need to extend them, and use **from** only where it is needed
- Consider these fragments:
  ...x: T; ... x.f(p) ...
  ...y: **from** T; ... y.f(p) ...
- **x.f(p)** is statically bound
  - The prover can expand the call using the result expression defined in the declaration of **f**.
- **y.f(p)** is dynamically bound
  - unless **f** is declared **final** in class **T** or one of its ancestors
  - the prover cannot expand the call unless the precise type of **y** is known at the point of call.
  - the best that the prover can do is to assume any postassertion that was provided in the declaration of **f** applicable to class **T**.

Adding proof hints

- Any assertion (whether an embedded assertion or a post-assertion) can have a proof list attached like this:
  assert e1 **proof** p1; p2; ... pn **end**
- Each element in the proof list must be one of:
  - an assertion
  - a let-declaration
  - (final element only) a conditional proof - like a conditional statement, but each branch is a guarded proof list instead of a guarded statement list
- The assertions in the proof list are treated as lemmas
- The prover tries to prove the lemmas, then the main assertion