Formal Methods in Software Development
Exercise 6 (December 21)

Wolfgang Schreiner
Wolfgang.Schreiner@risc.uni-linz.ac.at

November 26, 2009

The result is to be submitted by the deadline stated above via the Moodle interface as a .zip or .tgz file which contains

- A PDF file with
  - a cover page with the title of the course, your name, Matrikelnummer, and email-address,
  - for each exercise, a section with the number and name of the exercise, the JML-annotated Java code, a copy of the output of an `jml -Q` and an `escjava2` check of that code,
  - optionally any explanations or comments you would like to make;
- the JML-annotated Java files developed in the exercise.

6a (50 points): A private JML Class Specification

Take the attached source code of a class `List` which implements a linked list of `Object` values and extend it by a *private* specification in the *heavy-weight* JML format that is as expressive as possible. Some hints:

- You can specify the `Node` constructor by a public specification (because all data fields are public).
- Formulate for `List` an object invariant which states that the sequence of linked nodes is not cyclic (use the given pure function (predicate) `isCyclic`) and not shared with the nodes of any other list (use the given predicate `isShared`).
- You may use the given pure function `getNode` for the specification of `getLastNode`. Investigate the specification of `getNode` by recursion over the natural number parameter i (two cases $i = 0$ and $i > 0$) and apply a similar strategy for the recursive specification of the private function `length` over the `Node` argument *head* (what are the two cases here?).
- Using the given private pure functions, specify the public constructor and the public functions.
- The two public functions `append` may modify the last node in the linked sequence. For these functions, it is in standard JML unfortunately not possibly to give adequate `assignable` clauses (some JML extensions allow to specify `assignable \reach(head)` to indicate that only the objects reachable via `head` can be specified). In these cases, you have to use `assignable \everything`.

Use `jml -Q` to check the specification (which must not yield an error). Run `escjava2` on the specification; even with the best possible JML specification, the tool will complain about possibly broken postconditions and object invariants. Check the warnings, and *if you are very confident* that everything is fine, insert the *minimal* set of `//@nowarn Post` (respectively `Invariant` or `Post, Invariant`) annotations required to switch off these warnings.

The result of this exercise contains the JML-annotated file `List.java` and the output of `jml -Q` and `escjava2` on this file (in the last case, once without and once with the `nowarn` annotations).

6b (50 points): A model-based public JML Class Specification

Take the previously JML-annotated file `List.java` and modify it for an appropriate *public* specification of class `List`; this public specification is to be written into file `List.jml` and shall be based on the abstract datatype `ListModel` specified in the attached file `ListModel.java`. Some hints:

- The basic strategy is the same as shown in class for the model-based public specification of class `IntStack`.
- Introduce in `List.jml` a model field of type `ListModel` which receives its value from a model function `toModel()`.
- Give in `List.jml` public specifications of the public functions using the model field and the corresponding operations on `ListModel`.
- Annotate `List.java` by a `refines` annotation that indicates that the definition of class `List` in this file is a refinement of the class declared in `List.jml`. Add the keyword `also` to the private behavior specifications of all public methods.
- Give a specification-only definition of the abstraction function `toModel` as

```
/*@ public pure model ListModel toModel() {
  @   ListModel m = new ListModel();
  @   Node node = head;
  @   while (node != null)
  @   {
  @     m = m.append(node.value);
  @     node = node.next;
  @   }
  @   return m;
  @ }
  @*/
```

  Annotate this definition with a *private* behavior specification that relates the constructed `ListModel` to the current `List` object (this is easily possible using the functions `length` and `get` in both types).
- Add the private variable `head` to the data group of the model variable; thus whenever an assignment on the model variable in the public specification is allowed, also an assignment to `head` in the implementation is allowed.
- Nevertheless, in the two `append` functions, the assignments `last.next = node` are not allowed (there is in standard JML no way to add all nodes of a linked list to a data group). Annotate the two assignments by `//@ nowarn Modifies` to get rid of corresponding warnings.

First use `jml -Q` to type-check `List.jml` in a directory that contains also `ListModel.java` but does *not* contain `List.java` (otherwise also this file will be immediately type-checked). As soon as the type-check succeeds, also add `List.java` from the previous exercise to this directory and extend it as indicated above.

Now use `jml -Q` again to type-check the files. As soon as everything is fine, try `escjava2` which may again complain about possibly invalid post-conditions and invariants. Check these warnings; if you are confident that everything is fine, turn them off by `nowarn` annotations.

The exercise result contains the JML-annotated files `List.jml`, `List.java`, `ListModel.java`, and the output of `jml -Q` and `escjava2` (in the last case, once without and once with the `nowarn` annotations).