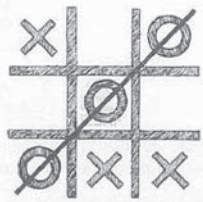
Program Verification (192114300)

Exam 25 June 2013, 13:45–17:15

- This exam contains 5 exercises, for which at most 90 points are awarded. The final mark equals the number of points obtained divided by 10, plus one.
- During the exam, the use of all materials is allowed.

Exercise 1 (*20 points*)

Consider the (in)famous *tic-tac-toe* game, in which two players in turn have to mark one of the previously unmarked fields of a 3×3 playing board with either X or O. Player X always starts; a player has won if he manages to put his marks on all the fields in a row (either horizontally, vertically or diagonally). For instance, the following board represents a situation where player O has won.



Design a graph representation of this game that allows an easy representation of moves and the detection of a winner.

a. *(4 points)* Give the type graph of your representation. Full points will only be awarded for a solution in that takes advantage of the symmetry of the board.

b. *(3 points)* Give a graph representation of the start state for a 2×2-board (instead of the usual 3×3-board).

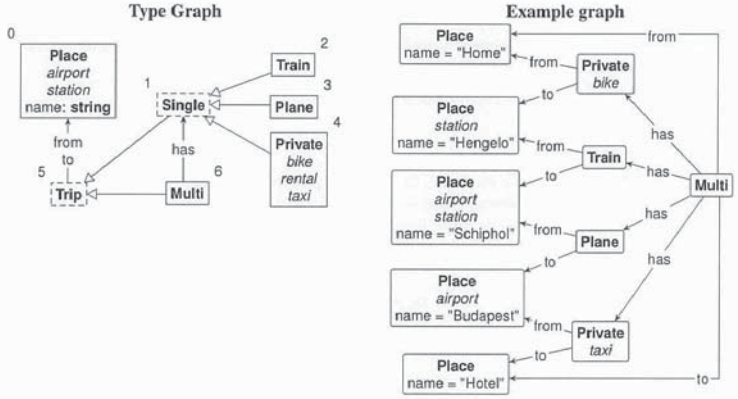
c. *(5 points)* Give one or more rules to represent a move. Full points will only be awarded for a solution in which all moves are applications of a *single* rule.

d. *(5 points)* Give one or more conditional rules to detect a winner. Full points will only be awarded for a solution in which all winning situations are applications of a *single* rule, which is, moreover, applicable both on the 2×2-board and on the 3×3-board.

e. *(3 points)* Draw the entire state space resulting for the 2×2-board, in which you collapse isomorphic graphs into single states. You do not have to draw the graphs of the individual states, but do make clear (by annotating your arrows) which transition corresponds to what (type of) move.

Exercise 2 (*20 points*)

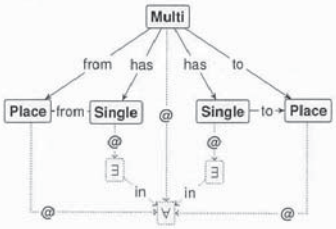
Consider the following type graph, which can be used to represent simple trips (consisting of a single leg with any of three types of transport) or composite trips (consisting of several consecutive legs). An example graph satisfying this type graph is also given.



The numbers in the type graph are node identities that will be used in the exercise below. Below, we call the target of a from-edge the *start place* of the trip, and the destination of the to-edge its *destination*.

a. *(6 points)* Is the following graph well-typed? If so, give a typing morphism that maps the graph nodes to the type nodes, and show that the morphism is correct; if not, argue that there does not exist such a morphism.

b. *(6 points)* What property does the following predicate graph express? Formulate the property in natural language and in first order logic.



c. *(8 points)* Express each of the following properties in first order logic and as graph predicates.

- A train trip must always start in a station.
- There is a leg of a composite trip starting in a place that is neither the start place of the composite trip nor the destination of any leg of that trip. *(If this property holds, if means that there is something wrong with the trip.)*

Exercise 3: Hoare Logic (*25 points*)

a. *(7 points)* Consider the statement $\texttt{pick}(S_1, S_2)$ that randomly chooses to execute either S_1 or S_2 (but not both). Its semantics is described by the following operational semantics rule:

$$\frac{(S_1, s) \to s' \vee (S_2, s) \to s'}{(\texttt{pick}(S_1, S_2), s) \to s'}$$

Give a Hoare logic rule for this statement, and argue why it is correct.

b. *(10 points)* Consider the statement $\texttt{doBound}(n)\{S\}$ that executes statement S exactly n times. Notice that the n just states how many times S should be executed, it is not a program variable that can be used in S. Its semantics is described by the following two operational semantics rules:

$$\frac{}{(\texttt{doBound}(0)\{S\}, s) \to s} \qquad \frac{(S, s) \to s' \quad (\texttt{doBound}(n-1)\{S\}, s') \to s''}{(\texttt{doBound}(n)\{S\}, s) \to s''}\, n > 0$$

Give a Hoare logic rule for this statement, and argue why it is correct. Since in a verification you might need to know how many times the body S has been executed, the rule should incorporate a special specification-only variable `iter` that denotes the number of times the body has been executed.

c. *(8 points)* Consider the method `approxVal`, using the statements defined above.

```
requires true;
ensures ?
method approxVal(int n) {
  int total := 0;
  doBound(n){
    pick{total := total + 1,
         total := total + 2};
  }
  return total;
}
```

Write a postcondition to specify the result value of `approxVal`, and use your rules to prove correctness of the method.

Exercise 4: Separation Logic (*25 points*)

Consider the class `MatrixList`. Its intention is to provide an alternative representation of lists: instead of storing data in a single list, it stores it in multiple smaller lists (this might for example allow a more efficient mapping to memory). However, the nodes stored in the `MatrixList` can be accessed as if it was a single list, using operations `get` and `set`.

```
class MatrixList {

  IntList thisList;
  MatrixList rest;

  // get node i from matrixlist
  List get(int i) {
    if (i < thisList.size()) {
      return thisList.get(i);
    } else {
      return rest.get(i - thisList.size());
   }
  }

  // set value in node in matrixlist
  void set(int i, int val) {
    if (i < thisList.size()) {
      thisList.set(i, val);
    } else {
      rest.set(i - thisList.size(), val);
    }
  }

  // return length of matrix list
  int size() {
    if (rest == null) {
      return thisList.size();
    } else
      return thisList.size() + rest.size();
    }
  }

}
```

For completeness, we give the declarations in class `List`, as used in `MatrixList`.

```
class List {

  // pred list<p,q,alpha> = ...

  int val;
  List next;

  // get node i from list
  List get(int i) {
    // ..
  }

  // set value in node i in list
  void set(int i, int val) {
  // ..
  }
```

```
  // return length of list
  int size() {
  // ..
  }
}
```

a. *(5 points)* Specify an abstract predicate `list` that captures that `MatrixList` represents a sequence α. You may assume that there is an appropriate `list` predicate defined for class `IntList` and that you have the necessary functions defined over sequences. The predicate should be usable in a multithreaded setting, thus it has to be parametrised with permissions. Make a distinction between the permission to change the list structure, and a permission to update the values.

b. *(3 points)* Write a specification for method `get` using this abstract predicate.

Consider now the method `sort` in Figure 1 that sorts the elements in a `MatrixList`. It does this by creating a large number of threads that in each round compare the value at position i with one of its neighbours, and if necessary swaps these (in fact, to sort N elements, $\lfloor N/2 \rfloor$ threads are created). By repeating this procedure sufficiently often (*i.e.*, at least N times), eventually the list will be sorted.

c. *(5 points)* Specify method `swap` for use in a multithreaded application, *i.e.*, using permissions.

d. *(6 points)* Specify a predicate `preFork` for method `run` in class `SortingThread` that is sufficient to prove that their will be no data races.

e. *(6 points)* The different rounds synchronise by means of a barrier. Give an appropriate specification for the behaviour of the barrier, and explain why this is sufficient to establish data race freedom of the complete `run` method.

```
class SortingThread {

  Barrier bar;
  int steps;
  int id;
  MatrixList sort;

  init (Barrier b, int s, int i, MatrixList m) {
    bar := b;
    steps := s;
    id := i;
    sort := m;
  }

  void swap(int id1, int id2) {
    int temp := sort.get(id1);
    sort.set(id1, sort.get(id2));
    sort.set(id2, temp);
  }

  void method run() {
    int count := 0;
    while (count < steps) {
      if (count % 0 == 0) {
        if (sort.get(id - 1) > sort.get(id)) {
          swap(id - 1, id);
        }
      } else {
        if (id + 1 < sort.size() & sort.get(id) > sort.get(id + 1)) {
          swap(id, id + 1);
        }
      }
      count := count + 1;
      bar.barrier();
    }
  }
}

method sort(MatrixList toSort) {
   int i := 1;
   int l := toSort.size();
   Barrier b := new Barrier(l/2);
   while (i < l) {
     SortThread tr := new SortThread;
     tr.init(b, l + 1, i, toSort);
     fork(tr);
     i := i + 2;
     // ...
   }
}
```

Figure 1: Method `sort`