CS476 Last Comprehensive Homework, Due at 5pm on Monday 5/11

Note: You should email by the above deadline:

- **four separate PDF files, one for each problem**,
- **for Problems 2 and 4**, the files with Maude code and the tool commands needed to solve those problems,

to `abir2@illinois.edu`. The time of Monday May 11 at 5pm is a **hard deadline**.
All tools and code necessary for completing the exam are available from the course webpage.
For reasonable tool-related questions/difficulties you can contact Michael Abir (`abir2@illinois.edu`).

1. Let $h : \mathcal{A} \longrightarrow \mathcal{B}$ be a Σ-isomorphism of order-sorted Σ-algebra, and $u = v$ a Σ-equation. Prove that

$$\mathcal{A} \models u = v \quad \Leftrightarrow \quad \mathcal{B} \models u = v$$

   For 50% extra credit. If you wish to earn 50% extra credit (that is, if this exercise is evaluated in a scale of 1 to 10 and you do everything right in the main part and in the extra credit part, you could get a total of 15 points), prove also that:
   $h : \mathcal{A} \longrightarrow \mathcal{B}$ be a Σ-isomorphism, then for any quantifier-free first-order formula φ we have the equivalence:

$$\mathcal{A} \models \varphi \quad \Leftrightarrow \quad \mathcal{B} \models \varphi$$

   Remarks. (1) This equivalence actually holds for any first-order formula φ with arbitrary nesting of quantifiers. The restriction to quantifier-free formulas is just to make your life easier in proving the equivalence. (2) The **main idea** proved by this interesting problem (and if you go for it by its extension) is that two isomorphic algebras are **identical**, up to a change of representation (think of numbers in binary or decimal notation), and therefore they satisfy the **same** properties, expressed here as formulas in equational (resp. first-order) logic.

2. Consider the definition of binary trees with quoted identifiers on the leaves given in Lecture 13. Complete the module given below (available in the course web page) by defining with confluent and terminating equations the two functions called `leaves` and `inner`, that count, respectively, the number of leaf nodes of a tree, and the number of nodes in a tree that are *not* leaf nodes. For example, for the tree `(('a # 'b) # 'c) # 'd` there are 4 leaf nodes (namely `'a`, `'b`, `'c`, and `'d`), and 3 inner nodes (corresponding to the 3 different occurrences of the `#` operator).

```
fmod TREE is
  protecting QID .
  sorts Natural Tree .
  subsort Qid < Tree .
  op 0 : -> Natural [ctor].
  op s : Natural -> Natural [ctor].
  op _+_ : Natural Natural -> Natural [assoc comm].
  op _#_ : Tree Tree -> Tree [ctor] .
  ops leaves inner : Tree -> Natural .
  var I : Qid .
  vars N M : Natural .
  vars T T' : Tree .
  eq N + 0 = N .
  eq N + s(M) = s(N + M) .
  *** add equations for leaves and inner here
endfm
```

Once you have defined and tested your definitions for `leaves` and `inner` do the following, *including screenshots for each tool used* in your hardcopy of the homework:

- check that it is sufficiently complete using the SCC tool
- state a theorem, in the form of a universally quantified equation, that gives a general law stating, for any tree `T`, the exact relation between the numbers `leaves(T)` and `inner(T)`
- give a mechanical proof of that theorem using the ITP

3. Fairness assumptions are very important to prove *liveness* properties about a system. Many liveness properties do not hold for *all behaviors* of the system, but only for *reasonable behaviors*, namely, fair ones.

   A very common notion of fairness is *transition fairness*, which for a rewrite theory can be understood as *rule fairness*, namely, for all rules with a common label l, rule fairness is the LTL formula $\mathit{fair}.l$, defined by the equality:

   $$\mathit{fair}.l = (\Box\Diamond\ \mathit{enabled}.l) \rightarrow (\Box\Diamond\ \mathit{taken}.l)$$

   which in plain English says that if l is infinitely often enabled to be fired, then l is infinitely often taken (fired).

   You have already seen examples of how the *enabled* state predicate can be defined in Lecture 18 (for all rules, but obviously this can be specialized to each rule label). Defining the $\mathit{taken}.l$ state predicate is slightly more tricky, since just looking at a state we may have no idea about which was the last rule applied. But it can be also easily done by *tagging* the state with the last rule label used. You saw an example of this tagging in the semantics of the `THREADED-IMP` language, and of how to express a fairness assumption (that both threads execute infinitely often) by using tagging in the model checking of liveness properties (non-starvation) for Peterson's algorithm in that lecture. There, what was tagged on the state was not quite the rule label, but the thread identifier of the last thread computing; but the idea is similar: we can instead tag the state with the label of the rule producing it.

   All this was by way of introduction to make clear that you *can* express rule fairness in your specifications. From now on let us assume you have done that. Let us come to the point about this problem: there is another very simple and closely related property for a rule l, namely the "leads to" property:

   $$\mathit{enabled}.l \rightsquigarrow \mathit{taken}.l$$

   Of course, the above "leads to" property is also closely related to the LTL formula:

   $$\mathit{enabled}.l \rightarrow \Diamond \mathit{taken}.l$$

   You are asked to answer the following questions about the relationships between: (a) the property $\mathit{fair}.l$, (b) the $\mathit{enabled}.l \rightsquigarrow \mathit{taken}.l$ property, and (c) the property $\mathit{enabled}.l \rightarrow \Diamond \mathit{taken}.l$: (i) are any of them semantically equivalent formulas? (ii) are some of them different? (iii) if different, does satisfaction of any of them imply satisfaction of some of the others? (iv) are any two of these properties *independent*, in the sense that none of the two implies the other?

   You should give a *mathematical proof* of your answers to (i)–(iv) in terms of the *semantics* of LTL. Some of your answers may take the form of *counterexamples*.

   Hint: Since LTL semantics is universally quantified over all the paths of a Kripke structure from an initial state, it may sometimes be useful to reason about what happens on a sigle path. For instance, to show that a given formula holds on that path but another formula does not; or that if one formula holds on the path, then another formula must also hold.

4. Consider the following program, which we will call *append-two*:

   $$\textbf{while}\ (i <: j)\ \{y =_l (1\ \$: 1)\ \$: y\ ;\ x =_l 1\ \$: x\ ;\ i = i +: 1\ ;\}$$

   This program appends two elements to the list stored in y and one element to the list stored x in every iteration, and increments the index i. We further assume that *in the initial state* the length of the list X stored in the variable x *equals* the length of the list Y stored in variable y.

   In this problem, you are expected to find an *inductive invariant* expressing an algebraic relation $\varphi(Z, X, Y)$

between the lengths of the lists held in x, y, and in an *auxiliary variable* z, which initially holds a list Z *having the same length* as that of the initial lists X and Y. You should think of Z in the auxiliary variable z as a *parameter* of this invariant.

Hint. Although other algebraic operations might be used, to help the IMPL Prover discharge your loop invariant automatically, we *strongly recommend* that your formula $\varphi(Z, X, Y)$ for the loop invariant only uses the $+$ and $*$ operations (could even use just $+$), as well as the defined *len* operation. You can always achieve this requirement by performing some *algebraic simplification* of the original formula you came up with. Once you have come up with the right invariant, you are asked to prove its correctness using the IMPL Prover.

To do this, we have provided the functional module that defines the length function on a list, which is called *len*. The following is a template for your proof program, in which you must fill in the two constraints in the *add-goal* section. The full template can be found on the course webpage, along with a corresponding Maude version and RLTool version. Using different versions of either may result in unexpected issues.

```
load impl.maude

fmod LEN is
  pr IMPL-LIST .

  op len : List -> Nat [metadata "90"] .

  var L : List . var N : Nat .

  eq len(nil) = 0 .
  eq len(N $ L) = 1 + len(L) .
endfm

mod LEN+IMPL-SYNTAX+MUL is
  pr LEN .
  pr IMPL-SYNTAX+MUL .
endm

mod LEN+IMPL-SEMANTICS is
  pr LEN .
  pr IMPL-SEMANTICS .
endm

set show advisories off .

load rltool.maude

(select LEN+IMPL-SEMANTICS  .)
(use tool conrew for validity on LEN+IMPL-SYNTAX+MUL
                 with FOFORMSIMPLIFY-IMP-IMPL .)
(use tool varunif for varunif on FVP-NAT .)
(use tool varsat  for unsatisfiability on IMPL-SYNTAX .)
(def-term-set (< done | St:Store >) | true .)
(declare-vars (X:List) U (X':List) U (Y:List) U (Y':List) U (Z:List) U (Z':List) U
              (I:Nat) U (I':Nat) U (J:Nat) U (J':Nat) U (K:Continuation) .)

(add-goal length :
             (<  while (i <: j) { y =l (1 $: 1) $: y ; x =l 1 $: x ; i = i +: 1 ; } ~> K
                | (x |-> TList * y |-> TList * z |-> TList * i |-> TNat * j |-> TNat)
                & (x |-> X * y |-> Y * z |-> Z * i |-> I * j |-> J) >)
                | *** Precondition Constraint Here *** =>
```

```
            (< K
               | (x |-> TList * y |-> TList * z |-> TList * i |-> TNat * j |-> TNat)
               & (x |-> X' * y |-> Y' * z |-> Z' * i |-> I' * j |-> J') >)
               | *** Midcondition Contraint Here *** .)

(start-proof .)
```