CS 476 Homework #5 Due 10:45am on 9/29

Note: Answers to the exercises listed below in *typewritten form* (latex formatting preferred) as well as code solutions should be emailed by the above deadline to `nishant2@illinois.edu`.

1. In the slides for Lecture 6, given an equational theory (Σ, E) the *joinability relation* $t \downarrow_{\vec{E}} t'$ is defined by the equivalence:
$$t \downarrow_{\vec{E}} t' \quad \Leftrightarrow \quad (\exists w)\ (t \rightarrow^*_{\vec{E}} w \wedge t' \rightarrow^*_{\vec{E}} w).$$
That is, $t \downarrow_{\vec{E}} t'$ holds iff t and t' *can both be rewritten to a common term* w (the more general modulo B case for $t \downarrow_{\vec{E}/B} t'$ is defined in Lecture 6 in the same way; but in this exercise we will stick to the case without axioms $(B = \emptyset)$).

   Prove the *Church-Rosser Theorem* (also stated in Lecture 5) in the following form:

   If (Σ, E) is a kind-complete order-sorted equational theory such that the rules $\vec{E}$ are confluent, then for any two Σ-terms t, t' whose sorts are in the same connected component we have the equivalence:
$$t =_E t' \quad \Leftrightarrow \quad t \downarrow_{\vec{E}} t'.$$
Hint: Use induction on the length of the proof of $t =_E t'$.

2. This problem is a good example of the motto:

   Declarative Maude Programming = Computable Mathematical Modeling

   In this case, the goal is to define a mathematical model of the set $\mathcal{P}_{fin}(\mathbb{N})$ of *finite* subsets of the set $\mathbb{N}$ of natural numbers in Peano notation and some commonly used set-theoretic functions on that powerset.

   Since your *mathematical model* should be the canonical term algebra $\mathbb{C}_{\Sigma/E,B}$ of the functional module `fmod` $(\Sigma, E \cup B)$ `endfm` that you will specify, you should *not use any built-in features* of Maude. In particular,

   No use should be made of the built-in equality predicate `==` *in any equations.*

   The built-in equality predicate `==` is very convenient. But by using it you are not giving a full mathematical definition. Here you are asked to give a *full mathematical definition* of all the functions involved in an algebra of (finite) sets of natural numbers, which should at the same time be a *correct program* to compute all those operations in such an algebra.

   Finite sets of natural numbers are defined as expected, using a binary associative and commutative set union constructor $\_,\_$ with `mt` (the empty set) as its identity element. Since our model is of sets and *not* of multisets, there is a set idempotency equation —restricted to numbers to avoid non-termination: see Lecture 5— already provided for you in the skeleton below. You are asked to write equations defining the following functions ($\_ \cup \_$ has already been defined for you in the template below by the `assoc`, `comm` and `id: mt` axioms, and the idempotency equation):

   (a) $\_ \cup \_$ set union
   (b) $\_ = \_$ (as predicate on $\mathbb{N}$)
   (c) $\_ \in \_$ (membership predicate)
   (d) $\_ \setminus \_$ (set difference, written $\_ - \_$ in *STACS*)
   (e) $\_ \subseteq \_$ (subset predicate)

(f) $\_ = \_$ (as predicate on $\mathcal{P}_{fin}(\mathbb{N})$)

(g) $\_ \cap \_$ set intersection

(h) $|\_|$ cardinality of a (finite) set.

Some example tests are included for your convenience, and you should further check the correctness of your function definitions with other tests.

Hints:

- The built-in module `NAT` is included for your convenience because: (i) it supports decimal notation and also Peano notation: 3 can be written both as `3` and as `s(s(s(0)))`, which is very convenient: you can, for example, define the equality predicate between naturals just using the Peano notation; (ii) it imports the `BOOL` module, so you have at your disposal all the Boolean operations, which can be useful when defining some of the predicates; and (iii) `BOOL` itself imports the if-then-else-fi operator: this, again, can be helpful when defining some functions.
- The order in which the functions are introduced gives you a hint that some functions earlier in the list may be useful as auxiliary functions for defining other functions later down the list.
- Programming modulo axioms such as associativity, commutativity and identity is very powerful and allows writing very short programs. For example, the eight functions in this example can be defined with just 18 equations. However, with this power come also some risks: (1) some equations may be *non-terminating* due to unexpected applications of the `id:` axiom for $\emptyset$ as identity for $\_\cup\_$, (2) losing *sufficient completeness*: you may forget some cases in your equations if you are not careful, or, even worse, (3) *loss of confluence*, so that a function may have *two or more* different results, depending on the order in which equations are applied, and therefore *is not a function at all!*

This last risk (3) is the highest: if you rely on the order in which you have written your equations, there is a pretty good chance that you may have written some nonsense and you have not defined a function at all. This last error can be quite nasty, since *you may not be able to detect it by testing*: Maude's `red` command follows a fixed evaluation strategy, so you will only see *one* result for a test. Therefore, you have to ask yourself: *could I get a different result if the equations are applied in a different order?* For example, you may write your equations implicitly assuming that the sets a function is applied to have no repeated elements (i.e., implicitly assuming that the equation `N,N = N` has already been applied exhaustively to the function's arguments). Then, you may happily proceed to write nonsense equations that will give you different values for the same input, depending on how they are applied; but you may *not notice* this fatal mistake just by testing. The tests cases included below include evaluations of functions whose arguments are sets with repeated elements. They are meant not just as test cases, but also to help you check your equational definitions "in your head," that is, by checking on a piece of paper if some weird order of applying the equations *could* lead to a different result.

Note that, although this is a relatively simple example, the *standard of quality in programming* is a high one: you are supposed to define *exactly* the mathematical model of finite sets of natural numbers —as would be defined in, say, the *STAC* notes— just by writing a handful of equations E, so that the canonical term algebra $\mathbb{C}_{\Sigma/E,B}$ for your module, up to the slight (bijective) change of representation:

- $\emptyset$ versus `mt`
- $\{n_1, \ldots, n_k\}$ versus $n_1, \ldots, n_k$

exactly defines the algebra of finite sets of natural numbers, with sorts `Nat` and `Set` respectively interpreted as $\mathbb{N}$ and $\mathcal{P}_{fin}(\mathbb{N})$, and $\_{\mathbb{C}_{\Sigma/E,B}}$ interpreting the module's function *symbols* (in exactly that order) as the *function*s (a)–(h) mentioned above, and having the mathematical meaning exactly defined for them in *STACS*.

```
fmod SET-ALGEBRA is
  protecting NAT .
  sort Set .
  subsort Nat < Set .
```

```
  op mt : -> Set [ctor] .                              *** empty set
  op _,_ : Set Set -> Set [ctor assoc comm id: mt] .   *** set union

  vars N M : Nat .  vars U V W : Set .

  eq N,N = N .    *** set idempotency

  op _~_  : Nat Nat -> Bool [comm] .    *** equality predicate on naturals

  *** write your equations here, using Peano notation, i.e., 0 and s(N)

   op _in_  : Nat Set -> Bool .            *** set membership

  *** write your equations here

   op _\_  : Set Set -> Set .           *** set difference

  *** write your equations here

  op _C=_ : Set Set -> Bool .           *** set containment

  *** write your equations here

  op _~_  : Set Set -> Bool [comm] .    *** equality predicate on sets

  *** write your equations here

  op _/\_ : Set Set -> Set .           *** set intersection

  *** write your equations here

  op |_| : Set -> Nat .                  *** cardinality function

  *** write your equations here

endfm

*** all tests below should come out "true"

red 5 ~ 12 == false .
red 15 ~ 15 == true .

red 4 in (3,3,4,4,7) == true .

red 9 in (3,3,4,4,7) == false .

red (3,3,4,4,4,2,2,9) \ (3,3,3,4,2,7) == 9 .

red (4,4,4,2,2,7) \ (3,3,3,4,2,7) == mt .

red (3,3,4,4,4,2,2,9) C= (3,3,3,4,2,7) == false .

red (3,3,4,4,2,2,9,9) C= (3,4,2,7,9) == true .

red (3,3,4,4,4,2,2,7) ~ (3,3,3,4,2,7) == true .
```

```
red (3,3,3,4,2,2,7) ~ (3,3,3,4,2,2) == false .

red (3,3,3,4,4,4,2,2,7,9) /\ (3,3,4,4,2,7,7,1) == 2,3,4,7 .

red | 3,3,4,4,4,2,2,9 | == 4 .
```

You can retrieve this module as a "skeleton" on which to give your answer from the course web page. Also, send a file with your module and tests to `nishant2@illinois.edu`.