FDD3024: Module 1 Homework Problem Set

Mohammad Ahmadpanah and Karl Palmskog

Send your individually written problem solutions as a single PDF file by email to `smmah@kth.se`, at the latest on May 7, 2026 at 18:00. Collaboration between students and AI use is allowed, but you must write up your own solutions (handwriting allowed). Solutions are graded pass/fail based on demonstrated effort to solve the problems; correctness is not necessary.

1. Inductive Relations and Induction Proofs

Consider the following (inductive) definition of regular expressions:

$$r ::= \epsilon \mid c \mid r + r \mid r \cdot r \mid r^*$$

We recursively define a relation L which expresses when a regular expression r "matches" a string, defined as a list of characters c. We use the utility function `concat` on lists-of-lists that concatenates all element lists, where $\texttt{concat}([]) = []$ for the empty list $[]$.

$$\begin{aligned}
L(\epsilon, w) &:= w = [] \\
L(c, w) &:= w = [c] \\
L(r_1 + r_2, w) &:= L(r_1, w) \vee L(r_2, w) \\
L(r_1 \cdot r_2, w) &:= \exists w_1, \exists w_2, w = w_1 w_2 \wedge L(r_1, w_1) \wedge L(r_2, w_2) \\
L(r^*, w) &:= \exists s, \texttt{concat}(s) = w \wedge \forall w' \in s, L(r, w')
\end{aligned}$$

We next define a relation M using inductive rules that is meant to be equivalent to L:

$$\text{EPS}\ \frac{}{M(\epsilon, [])} \qquad \text{CHAR}\ \frac{}{M(c, [c])} \qquad \text{ALTL}\ \frac{M(r_1, w)}{M(r_1 + r_2, w)} \qquad \text{ALTR}\ \frac{M(r_2, w)}{M(r_1 + r_2, w)}$$

$$\text{SEQ}\ \frac{M(r_1, w_1) \quad M(r_2, w_2)}{M(r_1 \cdot r_2, w_1 w_2)} \qquad \text{STAR}\ \frac{}{M(r^*, [])} \qquad \text{STARS}\ \frac{M(r, w_1) \qquad M(r^*, w_2)}{M(r^*, w_1 w_2)}$$

a) Set up a proof by structural induction that for all r and w, $L(r, w)$ implies $M(r, w)$. List carefully all cases and their induction hypotheses. Sketch the reasoning to prove each case.

b) Set up a proof by structural induction that for all r and w, $M(r, w)$ implies $L(r, w)$. List carefully all cases and their induction hypotheses. Sketch the reasoning to prove each case.

2. Binary Tree Extension

Consider a binary tree as a data type that does not store any values. In CakeML, this can be defined as follows:

```
datatype btree = Leaf | Branch btree btree
```

a) Follow Harper's approach with natural numbers from PFPL Chapter 9 and define the syntax for the T language extended with binary trees, including syntax for trees themselves (and tree types) and a recursor for trees.

b) Provide statics for your extended T language, i.e., provide a typing judgment/relation.

c) Provide dynamics for your extended T language, i.e., provide a reduction judgment/relation.

d) Specify and sketch a proof of safety for your extended language, similar to Harper's Theorem 9.3.

3. Integer Set Abstract Datatype

Consider an abstract datatype `t` representing a set of elements of a built-in integer type `int`, with the following three operations:

- `empty : t`
- `add : t -> int -> t`
- `contains : t -> int -> bool`

Consider the datatype `btree` of binary trees with `int` elements, in CakeML:

```
datatype btree = Leaf | Branch btree int btree
```

a) Fully write out an obviously correct implementation of the abstract type of sets and its operations using a single (polymorphic) list. Use a syntax similar to the ML family (e.g., CakeML, OCaml, or Standard ML).

b) A `btree` is a *binary search tree* (BST) when, for each branch in the tree, the integer at that branch is greater than every integer in the branch's left subtree, and less than each integer in the branch's right subtree. Give inductive judgments/rules that specify when a `btree` is a BST.

c) Define a `btree` instance with at least 6 `Leaf` constructors that is a BST. Show that the `btree` is indeed a BST by giving a formal proof tree using your rules.

c) Implement the integer set abstract datatype using `btree`, while ensuring that:

- `empty` is a BST (according to your definition), and
- if the binary tree argument to `add` is a BST, then it returns a binary tree that is a BST.

You can assume for `contains` that the given binary tree is a BST.

d) State the requirements for a relation R between the obviously correct implementation and your binary tree implementation to be a bisimulation relation, as per PFPL Chapter 17.4.

e) Find a bisimulation relation R that relates the obviously correct integer set implementation and your binary tree implementation as per PFPL Chapter 17.4, and sketch the argument that it fulfills the requirements.

4. Recursive Datatypes and Functions

Consider Harper's natural numbers from PFPL Chapter 9. For all tasks requiring definitions, use a syntax similar to the ML family (e.g., CakeML, OCaml, or Standard ML).

a) Encode natural numbers as an ML-style datatype.

b) Define addition as a recursive function `add` on the natural number datatype defined in a) using Harper's recursion equations from the note about Lambda calculus as a guide.

c) Define multiplication as a recursive function `mul` on the natural number datatype using the recursion equations above as a guide, calling the addition function defined before.

d) Define a "truncated subtraction" function taking two natural numbers n and m, returning 0 if n is less than or equal to m, and the usual $n - m$ otherwise.

e) Prove using structural induction on your natural number datatype that for all such numbers m and n,

$$m + (n - m) = m - n + n$$

using your above definitions of addition and truncated subtraction. Auxiliary lemmas may be needed.

5. Dependent Types

In a dependently-typed language, we can define a *sigma type* `sig A P`, with a type `A` and a predicate `P`, which represents the subset of elements (terms) in `A` that satisfy `P`. A dependently-typed language can also have predicate types as arguments types to functions, such as `Q x` for `x` an argument integer. That is, a function `f` could have type `forall (x : int). Q x -> sig A P`, sometimes written $\Pi\,(x : \texttt{int}).\, Q\,x \to \texttt{sig}\,A\,P$ in mathematical notation.

a) Explain briefly how predicate types can be used to express function preconditions to make it "partial". Exemplify by determining and writing out a precondition predicate type for the following function.

```
let isqrt (x : int) : int =
 let rec isqrt_aux (x : int) (n : int) : int =
   if x < (n + 1)*(n + 1) then n else isqrt_aux x (n + 1)
 in isqrt_aux x 0
```

b) Explain briefly how sigma types can be used to express function postconditions (ensures). Exemplify by writing a sigma type for the below function.

You can assume there is a relation `dupfree` that takes an element relation (of type `'a -> 'a -> bool`) and a list (of type `'a list`) and is true precisely when there are no duplicates in the list according to the element relation.

```
let rec undup (eq : 'a -> 'a -> bool) (l : 'a list) : 'a list =
 let rec inlb (eq : 'a -> 'a -> bool) (x : 'a) (l : 'a list) : bool =
  match l with
  | y :: l' -> if eq x y then true else inlb eq x l'
  | [] -> false
 in
 match l with
 | x :: l' -> if inlb eq x l' then undup eq l' else x :: undup eq l'
 | [] -> []
```

c) Write a strong, meaningful specification for the `gcd` function below entirely as a (dependent) type that uses predicate types and a sigma type.

```
let rec gcd (a : int) (b : int) : int =
  if a = 0 then 0
  else if b = 0 then 0
  else if a = b then a
  else if a < b then gcd a (b-a)
  else gcd (a-b) a
```