DD2552 semteo23: Homework Problem Set 2

Karl Palmskog

Please hand in your individually written solutions by 18:00, October 13, 2023 on Canvas or by email to palmskog@kth.se.

Solutions will be graded A-F. Problems are marked either E or C. All E problems must be solved with only minor errors to receive grade E or higher. C problems also have a number of points, and if you have solved all E problems with only minor errors, the total number of points will determine a grade A-E, as follows:

- 0 - 9 points: E
- 10 - 19 points: D
- 20 - 29 points: C
- 30 - 39 points: B
- 40 - 50 points: A

1. Purely Functional Integer Square Root

Assume there is a built-in integer type `int` like in WhyML and CakeML, and assume two expressions of this type can be compared by the operator `=`, as in `if n = 0 then 1 else 2`. An *integer square root* of a non-negative `int` value `x` is the greatest integer less than or equal to the (real-valued) square root of `x`. For example, the integer square root of 5 is 2.

a) [E] Propose a contract (requires and ensures) for a function `isqrt` computing the integer square root of an argument integer `x`.

b) [E] Write a purely functional implementation of `isqrt` that (you believe) upholds your function contract. You may use auxiliary functions, and there is no need to prove correctness or termination. Write the program in a syntax similar to the ML family (e.g., WhyML, CakeML, OCaml, or Standard ML). The function does not need to be efficient. You may use basic operators on `int` other than `=` such as `<=` and `<`.

c) [C, 5 points] Sketch a proof that your function upholds your contract.

d) [C, 5 points] Specify a termination measure as a `variant` declaration and sketch an argument how and why the measure ensures termination for all inputs that satisfy your requires declaration.

2. Contract for higher-order function

Consider the following function `f` defined using WhyML syntax:

```
let f (eq: 'a -> 'a -> bool) st h (v : 'b) nm =
 if eq nm h then v else st nm
```

a) [E] Describe briefly in words what the function does, assuming `eq` represents an equivalence relation.

b) [E] Rewrite the function with a more descriptive name while ensuring the types of all arguments and the return type are explicitly defined (i.e., provided as annotations inside the function definition).

c) [E] Propose a meaningful contract for the function in terms of requires and ensures. Be sure to cover both cases of the conditional.

d) [C, 5 points] Sketch a proof that your contract is upheld.

3. List duplication

Assume you are given a function `eq : 'a -> 'a -> bool` that represents a (decidable) equivalence relation.

a) [E] By using rules/judgments, define a relation called `in` on elements `x:'a` and lists `l:'a list` that represents list containment ("element is in the list") using `eq` for comparison.

b) [E] By using rules/judgments, define a predicate `dupfree` on lists that is true whenever the list has no duplicates according to `eq`.

c) [E] Write a structurally recursive function `undup` on lists that returns the argument list but without duplicates according to `eq`. It is recommended to use one or more auxiliary functions in the function definition.

d) [C, 5 points] Prove by structural induction that for all lists l, $\mathsf{dupfree}(\mathsf{undup}(l))$.

4. 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 also a concrete datatype `btree` of binary trees with `int` elements, in CakeML:

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

a) [E] 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., WhyML, CakeML, OCaml, or Standard ML).

b) [E] 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) [E] 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) [E] 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 that `contains` is always given a binary tree that is a BST.

d) [E] 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) [C, 8 points] Find a bisimulation relation R that relates the obviously correct integer set implementation and your binary tree implementation as per PFPL 17.4, and sketch the argument that it fulfills the requirements.

5. Optimizing depth-first search

We consider the following CakeML basis library function as a black box:

```
List.member: 'a -> 'a list -> bool
```

You can assume that this function always runs in linear time in the size of the input list. Then consider an implementation of depth-first search as a function `dfs` in CakeML:

```
fun fold_left f z s =
 case s of
   [] => z
 | x::s' => fold_left f (f z x) s';

fun dfs g n v x =
 if List.member x v then v else
 if n = 0 then v else fold_left (dfs g (n-1)) (x::v) (g x);
```

a) [E] Provide the type signatures of `fold_left` and `dfs` in the same form as given for `List.member` above. Briefly explain the meaning of each argument to `dfs` in words.

b) [E] Provide, and explain in terms of the underlying graph, the output value you get from the call `dfs (fn x => [x+1,x+3]) 5 [] 0`.

c) [C, 5 points] Determine the weakest conditions you can on `g`, `n`, and `x` such that the function call `dfs g n [] x` successfully terminates after having explored the whole graph.

d) [C, 7 points] How could the arguments `g` and `v` be represented (purely functionally) in order to achieve better asymptotic running time than a naive implementation with lists? Explain briefly and write out a new function and its signature, where, e.g., you assume you have access to a module with some efficiently implemented abstract type and its operations.

6. Module type and modules for groups

In abstract algebra, a *group* is a set of elements T where:

- There is a binary operation $*$ (`mul`) on elements in T that returns an element in T.
- There is a special unique element $e \in T$ (`one`).
- There is an inverse operation $^{-1}$ (`inv`) on elements in T that returns an element of T.

a) [E] Define a module type (signature) using the Standard ML or OCaml syntax that represents an interface to (an abstract type that represents) a group.

b) [E] Write a module that represents the additive group of natural numbers modulo m (i.e., a group using addition modulo m as its operation `mul`) which implements your module type. Fix some $m \geq 3$ and use the representation of natural numbers from Homework Problem Set 1 Problem 5, i.e., Harper's natural numbers as an ML-style datatype.

c) [C, 5 points] Lift the restriction to a fixed modulo in your module from b) by parameterizing your implementation of the additive group of natural numbers on a natural number m (assumed to be > 0) using functors.

7. Dependent types

In a dependently typed language, we can define a *sigma type* `sig A P`, with `A` a type and `P` a predicate, which represents the subset of elements 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. I.e., a function `f` could have type `forall (x : int). Q x -> sig A P`.

a) [E] Explain briefly how predicate types can be used to express function preconditions (requires). Exemplify by writing a predicate type based on your requires of the `isqrt` function in Problem 1.

b) [E] Explain briefly how sigma types can be used to express function postconditions (ensures). Exemplify by writing a sigma type for your `undup` function in Problem 3.

c) [C, 5 points] 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 function gcd (a 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
```