Computer Science 320 (Fall, 2009)
Concepts of Programming Languages

# Assignment 4: Inductive Proofs

**Out: Tuesday, October 6, 2009**
**Due: Friday, October 16, 2009**

In this assignment, you will assemble a few automatically verifiable inductive proofs about pure functional code. All your solutions should be defined within a file named `foldfiles.pf`. **File names are case sensitive.**

You will need to obtain the Haskell source files for the IBIS lightweight proof verifier. Once this is done, you can verify a proof script using a Haskell interpreter by loading the `Main` module into the interpreter and evaluating:

```
ibis "filepath/filename.pf"
```

where `filepath/filename.pf` is a relative path from the location of the source files. If you choose to compile the verifier using the provided `Makefile`, you can run it from the command line using the same syntax as above.

**Problem 1.** (50 pts)

Suppose you have a large collection of files spread across many computers. Each computer has its own processor and can run its own instance of a Haskell interpreter. You need to produce an aggregate analysis of *all* the files as defined by a Haskell function called `agg`. In other words, for all the files across all the computers, you must compute

```
f1 `agg` f2 `agg` ...  `agg` fN
```

You do not know anything about the function's implementation, but you do know that there is a Haskell value `id` that acts as both a left and right identity for the function `agg`, and `agg` itself is associative. All this information is summarized in the following proof script.

```
Introduce agg, id.

Assume \forall f.  f `agg` id = f
Assume \forall f. id `agg` f  = f

Assume \forall f,g,h. f `agg` (g `agg` h) = (f `agg` g) `agg` h
```

You decide that the best way to accomplish this task is to perform a `foldr` over all the files. However, you plan to distribute the workload over all the machines and then combine the results at the end. You observe that this will be equivalent to folding over all the files on a single machine only if for any pair of file lists `xs` and `ys`, it is the case that

```
foldr agg id (xs ++ ys) = (foldr agg id xs) `agg` (foldr agg id ys)
```

Thus, it is necessary to prove that this equation holds.

(a) A Haskell definition for `foldr` is provided below.

```
foldr f b (x:xs) = f x (foldr f b xs)
foldr f b []     = b
```

Add appropriate assumptions to the proof script that mathematically represent the equations that constitute the definition of `foldr`.

(b) A Haskell definition for the list "append" operator is provided below.

```
(x:xs) ++ ys = x:(xs ++ ys)
[]     ++ ys = ys
```

Add appropriate assumptions to the proof script that mathematically represent the equations that constitute this definition.

Now that all the assumptions describing the circumstances have been added to the proof script, it is possible to construct a proof of the desired mathematical statement. Because the statement is quantified universally over two lists, the proof must proceed by induction. In order to receive full credit, your solutions must contain no new assumptions and only automatically verifiable assertions.

(c) First, the statement must be shown to hold in the base case for an empty list. Add to your proof script and complete the following proof of the base case by replacing `...` with an appropriate sequence of true statements, separated by the conjunction operator.

```
Assert \forall ys.
    ...
 /\ (foldr agg id []) `agg` (foldr agg id ys) = foldr agg id ([] ++ ys)
```

(d) Next, it remains to prove that the statement holds for any recursive case. It is sufficient to show that given an inductive hypothesis about some list `xs`, it is possible to derive the same statement for a list `x:xs`. Complete the following proof of the inductive case.

```
Assert \forall x,xs,ys.
     (foldr agg id xs)     `agg` (foldr agg id ys) = foldr agg id (xs ++ ys)
 => ...
 /\  (foldr agg id (x:xs)) `agg` (foldr agg id ys) = foldr agg id ((x:xs) ++ ys)
```

**Problem 2.** (50 pts)

As before, solutions that can receive full credit must contain no new assumptions and only automatically verifiable assertions.

(a) Suppose you also want to prove that `[]` is a right identity for `++`. That is, for any `xs`,

```
xs ++ [] = xs
```

Add two assertions to your proof script that correspond to a proof of this statement. The first should be the base case, in which `xs` is `[]`. The second should be the inductive case, in which you assume that the statement holds for some `xs` and derive that it holds for some `x:xs`:

```
Assert \forall x,xs.
    xs ++ [] = xs
 => ...
 /\ (x:xs) ++ [] = x:xs
```

(b) Suppose we want to show that `++` is associative. This means that for any lists `xs`, `ys`, and `zs`, it is the case that

```
(xs ++ ys) ++ zs = xs ++ (ys ++ zs)
```

Any case in which `ys = []` or `zs = []` is trivial, since `[]` is both a left and right identity of `++`. Thus, it is sufficient to prove that this holds for any `xs` by induction over the structure of `xs`. Assemble two assertions that correspond to the base and inductive cases of a proof of this statement. You must use an approach that is analogous to the two previous proofs. You may *not* introduce any additional assumptions.