Embedded Domain Specific Languages with Dependent Types: Exercises

Edwin Brady
ecb10@st-andrews.ac.uk

DSL 2013, July 9th-20th 2013

You will find links to course material at `http://www.idris-lang.org/documentation/dsl-2013/`. In particular, you will find the code for these exercises in the `exercises/` subdirectory of the course code distribution at `http://www.idris-lang.org/courses/DSL2013/course_code.tgz`.

You should not expect to complete all of the exercises in the time available in the exercise sessions! There are a lot of questions, which you can take at your own pace. Please feel free to ask questions either during the exercise sessions, or later by email or on the `#idris` IRC channel on freenode.

Part 1: Basic Idris

1. Write a function `repeat : (n : Nat) -> a -> Vect a n` which constructs a vector of `n` copies of an item.

2. Consider the following functions over `Lists`:

```
take : Nat -> List a -> List a
drop : Nat -> List a -> List a
```

   (a) What are the types of the corresponding functions for `Vect`, `vtake` and `vdrop`?
   Hint: What are the invariants? i.e. how many items need to be in the vector in each case?

   (b) Implement `vtake` and `vdrop`

3. A matrix is a 2-dimensional vector, and could be defined as follows:

```
Matrix : Type -> Nat -> Nat -> Type
Matrix a n m = Vect (Vect a m) n
```

   (a) Using `repeat`, above, and `Vect.zipWith`, write a function which transposes a matrix.
   Hints: Remember to think carefully about its type first! `zipWith` for vectors is defined as follows:

```
zipWith : (a -> b -> c) -> Vect a n -> Vect b n -> Vect c n
zipWith f []      []      = []
zipWith f (x::xs) (y::ys) = f x y :: zipWith f xs ys
```

   (b) Write a function to multiply two matrices.

4. The following *view* describes a pair of numbers as a difference:

```
data Cmp : Nat -> Nat -> Type where
     cmpLT : (y : _) -> Cmp x (x + S y)
     cmpEQ : Cmp x x
     cmpGT : (x : _) -> Cmp (y + S x) y
```

(a) Write the function `cmp : (n : Nat) -> (m : Nat) -> Cmp n m` which proves that every pair of numbers can be expressed in this way.
Hint: recall `parity` from the lecture. You will find the `with` construct very useful!

(b) Assume you have a vector `xs : Vect a n`, where `n` is unknown. How could you use `cmp` to construct a suitable input to `vtake` and `vdrop` from `xs`?

5. Implement the following functions:

```
plus_nSm : (n : Nat) -> (m : Nat) -> n + S m = S (n + m)
plus_commutes : (n : Nat) -> (m : Nat) -> n + m = m + n
plus_assoc : (n : Nat) -> (m : Nat) -> (p : Nat) ->
             n + (m + p) = (n + m) + p
```

6. One way to define a reverse function for lists is as follows:

```
reverse : List a -> List a
reverse xs = revAcc [] xs where
  revAcc : List a -> List a -> List a
  revAcc acc [] = acc
  revAcc acc (x :: xs) = revAcc (x :: acc) xs
```

Write the equivalent function for vectors, `vect_reverse : Vect a n -> Vect a n`

Hint: You can use the same structure as the definition for `List`, but you will need to think carefully about the type for `revAcc`, and may need to do some theorem proving.

7. You are given the following definition of binary trees:

```
data Tree a = Leaf | Node (Tree a) a (Tree a)
```

Define a membership predicate `ElemTree` and a function `elemInTree` which calculates whether a value in in the tree, and a corresponding proof.

```
data ElemTree  : a -> Tree a -> Type where ...

elemInTree : DecEq a =>
             (x : a) -> (t : Tree a) -> Maybe (ElemTree x t)
```

Part 2: Embedded DSLs

8. Add a `let` binding construct to the `Expr` language, and extend the `interp` function and `dsl` notation to handle it.

9. In `L2-imp.idr` you will find a partially implemented imperative DSL. Implement the following missing functions:

(a) `update : HasType i G t -> Env G -> interpTy t -> Env G`, which updates the value stored at a particular position in an environment.

(b) `eval : Env G -> Expr G t -> interpTy t`, which evaluates an *expression*

(c) `interp : Env G -> Imp G t -> IO (interpTy t, Env G)`, which interprets a *program*, returning a value paired with an updated environment.

10. One example program is the following:

```
small : Imp [] TyUnit
small = Let (Val 42) (do Print (Var stop)
                         stop := Op (+) (Var stop) (Val 1)
                         Print (Var stop))
```

Using `dsl` notation, and any other syntax overloading you find useful, make it possible to write `small` as follows:

```
small : Imp [] TyUnit
small = imp (do let x = 42
                Print x
                x := x + 1
                Print x)
```

11. Extend `Imp` with a `for` loop construct. One possible type for this is:

```
For : Imp G i -> -- initialise
      Imp G TyBool -> -- test
      Imp G x -> -- increment
      Imp G t -> -- body
      Imp G TyUnit
```

Your implementation should allow the following program to be written, which outputs numbers from 1 to 10:

```
count : Imp [] TyUnit
count = imp (do let x = 0
                For (x := 0) (x < 10) (x := x + 1)
                    (Print (x + 1)))
```

Part 3: Effects

12. Reimplement `eval` and `interp` from the Lecture 2 exercises so that they use `Eff` annotated with appropriate effects, rather than the `IO` monad and an explicit environment.

13. Implement an `ENVIRONMENT` effect which allows system environment variables to be read, and program arguments to be retrieved. The signature could be:

```
data Environment : Effect where
     GetEnv  : String -> Environment () () String
     GetArgs : Environment () () (List String)

ENVIRONMENT : EFF
ENVIRONMENT = MkEff () Environment
```

Write a handler for (at least) `IO`.

Hint: You will find the `System` module useful. See `https://github.com/edwinb/Idris-dev/blob/master/lib/System.idr`

14. Effectful progams may wish to support logging. Define an effect signature and handler which supports adding logs, and reading all logs added so far.