Graded types exercises

Dominic Orchard

For Granule install instructions and information see `http://granule-project.github.io/splv23`

1 Declarative specification of type systems

Recall the typing rules of the linear λ-calculus:

$$\frac{}{x : A \vdash x : A}\text{VAR} \quad \frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.t : A \multimap B}\text{ABS} \quad \frac{\Gamma \vdash t_1 : A \multimap B \quad \Delta \vdash t_2 : A}{\Gamma, \Delta \vdash t_1\, t_2 : B}\text{APP}$$

where Γ, Δ is a partial operation, concatenating contexts only when they are disjoint, and we can implicitly re-order within Γ (implicit exchange).

1. Give a typing derivation for the judgement:

$$f : A \multimap B \multimap C \vdash \lambda x.\lambda y.f\, y\, x : B \multimap A \multimap C$$

2. Why is there no typing derivation for the following judgement?

$$f : A \multimap A \multimap B \vdash \lambda x.f\, x\, x : A \multimap B$$

3. (*Extension*): Consider an extension to the linear λ-calculus with a contraction rule:

$$\frac{\Gamma, x : A, y : A \vdash t : B}{\Gamma, z : A \vdash t[z/x][z/y] : B}\text{CONSTR}$$

(recall $t[z/x][z/y]$ means replace z for x and replace z for y inside of t).

This is known as *relevant logic* (or *relevance logic*): we don't allow variables to be discarded but they can be used any number of times. In relevant logic, give a derivation of the judgement:

$$f : A \multimap A \multimap B \vdash \lambda x.f\, x\, x : A \multimap B$$

2 Linear types and the non-linear modality

1. (Kicking the tires). Make a Granule source file called `exercises.gr` with a simple definition such as the following:

```
dub : Int -> Int
dub x = 2 * x
```

You can then load this into Granule's interactive mode (`grepl`) as follows:

```
$ grepl
Granule> :l exercises.gr
~/granule/exercises.gr, checked.
Granule> dup 42
84
```

2. In Granule, make the following functions typecheck by addition of the non-linear 'box' modality (written `a []`) only where needed, and add the relevant unboxing pattern matches at the value level.

```
const : forall {a b : Type} . a -> b -> a
const x y = x

ap : forall {a b c : Type}. (c -> a -> b) -> (c -> a) -> (c -> b)
ap f x ctx = f ctx (x ctx)
```

3. Consider the following definition:

```
import Bool

copyBool : Bool -> (Bool, Bool)
copyBool False = (False, False);
copyBool True  = (True, True)
```

Why does `copyBool` not violate linearity?

4. Write a well-typed "short-circuiting" version of `and` on `Bool`, i.e., where `and False x` need not inspect the value of `x`.

3 Graded modal types

1. Define a higher-order polymorphic function `twice` that takes a function and composes it with itself, e.g. in the lambda calculus `\f -> \x -> f (f x)`. Give a precise type explaining the reuse of the function parameter using the `Nat`-graded modality.

   Give an example of its usage, e.g., applying some integer function twice to an integer input.

2. Define a version of `twice` that is polymorphic in the semiring.

3. Using the `Cake` module, define a function `mange` that takes $n + m$ cakes, consumes n of them, leaving m left and n amounts of happiness.

   Hint: Check the library documentation[1] for the `Cake` module.

The following data type defines patient records with a public field (city) and a private field (name):

[1] `https://granule-project.github.io/docs`

```
data Patient where
  Patient : (city : String [Public]) -> (name : String [Private]) -> Patient
```

4. Write a function to fold over a vector of patients, of the following type, where only the public field can be reduced on:

   ```
   reducePublic : forall {a : Type, n : Nat}
                . (String -> a -> a) [0..n] -> a -> (Vec n Patient) -> a
   ```

   What happens if you try to leak the private data in this implementation?

4 Session types

Recall the interface for working with session types in Granule:

```
send        :  LChan (Send a p) -> a -> LChan p
recv        :  LChan (Recv a p) -> (a, LChan p)
forkLinear  :  (LChan p -> ())  -> LChan (Dual p)
close       :  LChan End -> ()
```

1. Write a function implementing the following type signature, using channels to receive a boolean and send the integer equivalent (0 or 1) back:

   ```
   boolToInt : LChan (Recv Bool (Send Int End)) -> ()
   ```

2. Write a client function that can interact with `boolToInt`, returning an integer.

3. Write a function `exampleSession` that forks `boolToInt` and connects this to your client program. Test the output.

4. (*Extension*): The `forkReplicate` primitive provides a way to create a replicated server. Check its type in `grepl` (or the documentation in the `Primitives` library) and use it to create a repeated version of `boolToInt`.

5 Uniqueness and mutation

1. Write a function converting a `Vec n Float` into a unique `FloatArray` using mutation to initialise the float array, i.e., define a function of type:

   ```
   toFloatArray : forall {n : Nat} . Vec n Float -> *FloatArray
   ```

2. The following code sums up the elements of an immutable array between two indices:

   ```
   sumFromTo : FloatArray ->  Int [] -> Int [] -> Float
   sumFromTo array [i] [n] =
      if i == n
      then let () = drop @FloatArray array in 0.0
   ```

```
    else
      let (x, a) = readFloatArrayI array i
      in  x + (sumFromTo a [i+1] [n])
```

Using this function, define `parSum : *FloatArray -> Float` that computes a data parallel sum of a unique float array by first sharing to an immutable array and then summing up two separate halves in parallel (see the `par` function from the `Parallel` module).

Test your solution using your definition of `toFloatArray`.

e.g. `parSum (toFloatArray (Cons 10.0 (Cons 20.0 (Cons 30.0 (Cons 40.0 Nil)))))` should return `100.0`.

6 Linear Haskell

Recall from the lectures that Haskell provides a graded-base type system via the *linear types* extension, which you can enable with the language pragma:

```
{-# LANGUAGE LinearTypes #-}
```

(*Homework*): Go back and implement the programs from Section 2 using this extension. You might like to try some of the simpler non-linear examples from the lectures as well.