Exercise 3.5

F142001

June 19, 2015

I have added the new module (Regular2Multirec) to the bottom of this file. I have also added a few methods (map∘, map∀ and mapId) to the Multirec module, which is why I edited the original file instead of creating a new one.

All those special symbols in Agda make this completely unreadable as a PDF, sorry...

First, the extra methods:

```
map? : {I : Set} {A B C : Indexed I} (D : Code I)
       {f : ? i ? B i ? C i} {g : ? i ? A i ? B i} {i : I} {x : ? D ? A i}
       ? map D f i (map D g i x) ? map D (? i ? (f i) ? (g i)) i x
map? U = refl
map? (I ' x) = refl
map? (! x) = refl
map? (c? ? c?) {x = inj? _} = cong inj? (map? c?)
map? (c? ? c?) {x = inj? _} = cong inj? (map? c?)
map? (c? ? c?) {x = _ , _} = cong? _,_ (map? c?) (map? c?)

map? : {I : Set} (C : Code I) {A B : Indexed I} {f g : ? i ? A i ? B i} ? (? i x ?
map? U p x x? = refl
map? (I ' j) p j ' x? = p j x?
map? (! j) p j ' x? = refl
map? (c? ? c?) p j (inj? x) = cong inj? (map? c? p j x)
map? (c? ? c?) p j (inj? y) = cong inj? (map? c? p j y)
map? (c? ? c?) p j (x , y) = cong? _,_ (map? c? p j x) (map? c? p j y)

mapId : {I : Set} {A : Indexed I} (C : Code I) ? {i : I} ? {x : ? C ? A i} ? map {
mapId U = refl
mapId (I ' _) = refl
mapId (! _) = refl
mapId (c? ? c?) {x = inj? _} = cong inj? (mapId c?)
mapId (c? ? c?) {x = inj? _} = cong inj? (mapId c?)
mapId (c? ? c?) {x = _ , _} = cong? _,_ (mapId c?) (mapId c?)
```

Then the new module:

```
module Regular2Multirex where
  -- STEP 1. embedding of regular codes into multirec codes:
  r2mrC : Regular.Code ? Multirec.Code ?
  r2mrC Regular.U          = Multirec.U
```

```
r2mrC Regular.I'        = Multirec.I' tt
r2mrC (c? Regular.? c?) = (r2mrC c?) Multirec.? (r2mrC c?)
r2mrC (c? Regular.? c?) = (r2mrC c?) Multirec.? (r2mrC c?)
-- Note: the ! constructor of multirec is just not used at all, because we don't r
-- Every regular code is transformed into a multirec code with an index from ? (w
-- This represents the fact that indices are not really relevant in this case.

-- STEP 2. Define the isomorphism between the interpretations in both systems, and
-- This consists of two directions, "from" (Regular -> Multirec) and "to" (Multire
fromRegular : {R : Set} ? (C : Regular.Code) ? (Regular.?_? C R) ? ((Multirec.?_?
fromRegular Regular.U              tt = tt
fromRegular Regular.I'              x = x
fromRegular (c? Regular.? c?) (inj? x) = inj? (fromRegular c? x)
fromRegular (c? Regular.? c?) (inj? x) = inj? (fromRegular c? x)
fromRegular (c? Regular.? c?) (x , y)  = fromRegular c? x , fromRegular c? y

from?Regular : (C : Regular.Code) ? Regular.? C ? Multirec.? (r2mrC C) tt
from?Regular c Regular.? x ? = Multirec.?_? (fromRegular c (Regular.map c (from?R

toRegular : {R : Set} ? (C : Regular.Code) ? ((Multirec.?_? (r2mrC C) (? X ? R)) t
toRegular Regular.U = id
toRegular Regular.I' = id
toRegular (c? Regular.? c?) = [_,_] (inj? ? (toRegular c?)) (inj? ? (toRegular c?)
toRegular (c? Regular.? c?) = <_,_> ((toRegular c?) ? proj?) ((toRegular c?) ? pro

to?Regular : (C : Regular.Code) ? Multirec.? (r2mrC C) tt ? Regular.? C
to?Regular c Multirec.? x ? = Regular.?_? (toRegular c (Multirec.map (r2mrC c) (?


-- STEP 3. Show that the functions just defined indeed do form an isomorphism. Th
-- show that to ? from = id, and that from ? to = id. Again, both for the regular
-- STEP 3a. to ? from = id
iso? : {R : Set} ? (C : Regular.Code) ? (x : Regular.?_? C R) ? toRegular C (fromR
iso? Regular.U tt            = refl
iso? Regular.I' _            = refl
iso? (c? Regular.? c?) (inj? x) = cong inj? (iso? c? x)
iso? (c? Regular.? c?) (inj? y) = cong inj? (iso? c? y)
iso? (c? Regular.? c?) (x , y)  = cong? _,_ (iso? c? x) (iso? c? y)

-- helper's lemma for iso??. I could have written R? R? : Set, because they are tr
-- by giving tt as an argument. But I think this with this definition the structur
lemma? : {R? R? : Multirec.Indexed ?} (C : Regular.Code)
       {f : (R? tt) ? (R? tt)} (x : ((Multirec.?_? (r2mrC C) (? X ? (R? tt))) tt
     ? toRegular C (Multirec.map (r2mrC C) (? i ? f) tt x) ? Regular.map C f (to
lemma? Regular.U _             = refl
lemma? Regular.I' _            = refl
lemma? (c? Regular.? c?) (inj? x) = cong inj? (lemma? c? x)
lemma? (c? Regular.? c?) (inj? y) = cong inj? (lemma? c? y)
lemma? (c? Regular.? c?) (x , y)  = cong? _,_ (lemma? c? x) (lemma? c? y)
```

```
open ?−Reasoning

iso?? : (C : Regular.Code) ? (x : Regular.? C) ? to?Regular C (from?Regular C x) ?
iso?? c Regular.? x ? = cong Regular.?_? $
     begin
        toRegular c (Multirec.map (r2mrC c) (? i ? to?Regular c) tt (fromRegular c
     ?? lemma? c _ ? −− take map outside
        Regular.map c (to?Regular c) (toRegular c (fromRegular c (Regular.map c (f
     ?? cong (Regular.map c (to?Regular c)) (iso? c _)? −−use regular iso? to make
        Regular.map c (to?Regular c) (Regular.map c (from?Regular c) x)
     ?? Regular.map? c ? −− composition of two maps (map the composition of the tw
        Regular.map c (to?Regular c ? from?Regular c) x
     ?? Regular.map? c (iso?? c) x ? −− recursion step (to?Regular ? from?Regular
        Regular.map c id x
     ?? Regular.mapId c ? −− map id does nothing
        x ?

−− STEP 3b. from ? to = id. Similar to 3a, but had to define map?, map?, and mapIo
−− They were already defined in the Regular module.
iso? : {R : Set} ? (C : Regular.Code) ? (x : (Multirec.?_? (r2mrC C) (? X ? R)) tt
iso? Regular.U tt                = refl
iso? Regular.I' _                = refl
iso? (c? Regular.? c?) (inj? x) = cong inj? (iso? c? x)
iso? (c? Regular.? c?) (inj? y) = cong inj? (iso? c? y)
iso? (c? Regular.? c?) (x , y)  = cong? _,_ (iso? c? x) (iso? c? y)

−− helper's lemma for iso??.
lemma? : {R? R? : Multirec.Indexed ?} (C : Regular.Code)
          {f : (R? tt) ? (R? tt)} (x : Regular.?_? C (R? tt))
        ? fromRegular C (Regular.map C f x) ? Multirec.map (r2mrC C) (? i ? f) tt (
lemma? Regular.U x = refl
lemma? Regular.I' x = refl
lemma? (c? Regular.? c?) (inj? x) = cong inj? (lemma? c? x)
lemma? (c? Regular.? c?) (inj? y) = cong inj? (lemma? c? y)
lemma? (c? Regular.? c?) (x , y) = cong? _,_ (lemma? c? x) (lemma? c? y)

iso?? : (C : Regular.Code) ? (x : Multirec.? (r2mrC C) tt) ? from?Regular C (to?R
iso?? c Multirec.? x ? = cong Multirec.?_? $
   begin
      fromRegular c (Regular.map c (from?Regular c) (toRegular c (Multirec.map (r2
   ?? lemma? c _ ?
      Multirec.map (r2mrC c) (? i ? from?Regular c) tt (fromRegular c (toRegular c
   ?? cong (Multirec.map (r2mrC c) (? i ? from?Regular c) tt) (iso? c _) ?
      Multirec.map (r2mrC c) (? i ? from?Regular c) tt (Multirec.map (r2mrC c) (?
   ?? Multirec.map? (r2mrC c) ?
      Multirec.map (r2mrC c) (? i ? (from?Regular c ? to?Regular c)) tt x
   ?? Multirec.map? (r2mrC c) (? i ? iso?? c) tt x ?
      Multirec.map (r2mrC c) (? i ? id) tt x
   ?? Multirec.mapId (r2mrC c) ?
      (x ?)
```