Haskell Examples for Iteration and Coiteration on Higher-Order Datatypes

Andreas Abel*
Theoretical Computer Science, University of Munich
abel@informatik.uni-muenchen.de

January 31, 2003

This document gives a Haskell implementations of the examples in the article *Iteration and Coiteration for Higher-Order Nested Datatypes* by Abel and Matthes [AM03]. The programs make essential use of the rank-2 extensions of Haskell 98 and can be run under `hugs -98` or compiled with `ghc -fglasgow-exts`.

Thanks to Ralph Hinze who provided `lhs2TeX` which was used to type-set the literate Haskell sources automatically.

1 (Co)inductive types

```
module Rank1 where
```

Inductive types.

```
data Mu1 f = In1 (f (Mu1 f))
it1 :: Functor f => (f a -> a) -> Mu1 f -> a
it1 s (In1 t) = s (fmap (it1 s   )    t)
```

Coinductive types.

```
data Nu1 f = forall a. Coit1 (a -> f a) a
out1 :: Functor f => Nu1 f -> f (Nu1 f)
out1 (Coit1 s t) = fmap (Coit1 s) (s t)
```

2 (Co)inductive functors

```
module Rank2 (Functor2, Mu2, In, it, Nu2, Coit, out) where
```

*Supported by the PhD Programme *Logic in Computer Science* (GKLI) of the *Deutsche Forschungsgemeinschaft*.

-- module Rank2 (Functor2, Mu2, it, Nu2, out) where

Rank 2 monotonicity.

```
class Functor2 ff where
    ffmap :: (forall a b. (a → b) → f a → g b) →
             (a → b) → ff f a → ff g b
```

Inductive functors. Formation and introduction.

```
data Mu2 ff a = In (ff (Mu2 ff   a)
```

Elimination and computation.

```
it :: Functor2 ff ⇒ (forall b    .   ff g b → g b) →
    (a → b) → Mu2 ff a → g b
it s f (In t) = s (ffmap (it s    )   f t)
```

Functoriality.

```
instance Functor2 ff ⇒ Functor (Mu2 ff) where
    fmap = it In
```

Coinductive functors. Formation and introduction.

```
data Nu2 ff b = forall g a. Coit
    (forall a. g a → ff g a)
    (a → b)
    (g a)
```

Elimination and computation.

```
out :: Functor2 ff ⇒ Nu2 ff a → ff (Nu2 ff) a
out (Coit s f t) = ffmap (Coit s) f (s t)
```

Functoriality.

```
instance Functor2 ff ⇒ Functor (Nu2 ff) where
    fmap = Coit out
```

3 Powerlists

3.1 Powerlist reversal

First, the version of powerlists where the map operation performs a reversal of the whole list.

```
module RevPowerlist where
import Prelude hiding (succ)
import Rank2 (Functor2, Mu2, In, it, Nu2, Coit, out)
```

The constructor of pure kind 2 which we need to define powerlists. We take the freedom and use labelled sums.

```
data PListF f a = Zero a | Succ (f (a, a))
```

We give a monotonicity witness which performs reversal.

```
swap :: (a → b) → (a, a) → (b, b)
swap f (a1, a2) = (f a2, f a1)
pListFRev :: (forall a b. (a → b) → f a → g b) →
                (a → b) → PListF f a → PListF g b
pListFRev s f (Zero a) = Zero (   f a)
pListFRev s f (Succ l) = Succ (   s (swap f) l)

instance Functor2 PListF where
    ffmap s f l = pListFRev s f l
```

Inductive type of powerlists.

```
type PList a = Mu2 PListF a
```

Powerlist constructors.

```
zero :: a → PList a
zero a = In id (Zero a)
succ :: PList (a, a) → PList a
succ l = In id (Succ l)
```

Fast powerlist reversal.

This is just the map function.

```
pListRev :: PList a → PList a
pListRev = fmap id
```

3.2 Powerlist summation

```
module Powerlist where
import Prelude hiding (succ, sum)
import Rank2 (Functor2, Mu2, In, it, Nu2, Coit, out)
```

```
pair :: (a → b) → (a, a) → (b, b)
pair f (a1, a2) = (f a1, f a2)
```

The constructor of pure kind 2 which we need to define powerlists. We take the freedom and use labelled sums.

```
data PListF f a = Zero a | Succ (f (a, a))
instance Functor2 PListF where
    ffmap s f (Zero a) = Zero (   f a)
    ffmap s f (Succ l) = Succ (   s (pair f) l)
```

Inductive type of powerlists.

```
type PList a = Mu2 PListF a
```

Powerlist constructors.

```
zero :: a → PList a
zero a = In (Zero a)
succ :: PList (a, a) → PList a
succ l = In (Succ l)
```

Summing up a powerlist of Integers.

We make use of the right Kan extension. Unfortunately, we cannot use a type definition and need a datatype instead.

```
newtype RKanInt a = RKanInt ((a → Integer) → Integer)
```

Step term.

```
stepSum :: PListF RKanInt a → RKanInt a
stepSum (Zero a) = RKanInt (λf → f a)
stepSum (Succ (RKanInt l)) = RKanInt (λf → l (λ(a1, a2) → f a1 + f a2))

sum' :: (a → b) → PList a → RKanInt b
sum' = it stepSum

sum :: PList Integer → Integer
sum l = k id
    where (RKanInt k) = sum' id l
```

4 De Bruijn terms

```
module DeBruijn where
import Prelude hiding (abs)
import Rank2
```

Rank 2 type constructor for a de Bruijn representation of lambda-terms.

```
data LamF f a = Var a | App (f a) (f a) | Abs (f (Maybe a))
instance Functor2 LamF where
    ffmap s f (Var a) = Var (f a)
    ffmap s f (App t1 t2) = App (s f t1) (s f t2)
    ffmap s f (Abs r) = Abs (s (fmap f) r)
```

Type of de Bruijn terms over a variable set A.

```
type Lam a = Mu2 LamF a
```

De Bruijn term constructors.

```
var :: a → Lam a
var a = In (Var a)
app :: Lam a → Lam a → Lam a
app t1 t2 = In (App t1 t2)
abs :: Lam (Maybe a) → Lam a
abs r = In (Abs r)
```

Weakening.

```
weak :: Lam a → Lam (Maybe a)
weak t = fmap Just t
```

Parallel substitution.

Step term.

```
newtype RKanLam b = RKanLam (forall c. ((b → Lam c) → Lam c))
stepSub :: LamF RKanLam a → RKanLam a
stepSub (Var a) = RKanLam (λsigma → sigma a)
stepSub (App (RKanLam t1) (RKanLam t2)) = RKanLam (λsigma →
    app (t1 sigma) (t2 sigma))
stepSub (Abs (RKanLam r)) = RKanLam (λsigma →
    abs (r (maybe (var Nothing) (   weak. sigma))))
```

Substitution in general form.

```
subst' :: (a → b) → Lam a → RKanLam b
subst' = it stepSub
```

Substitution (monad operation).

```
subst :: Lam a → (a → Lam b) → Lam b
subst t sigma = k sigma
    where (RKanLam k) = subst′ id t
```

Join operation.

```
join′ :: (a → b) → Lam (Lam a) → Lam b
join′ f t = k id where (RKanLam k) = subst′ (fmap f) t

join :: Lam (Lam a) → Lam a
join t = subst t id
```

5 Functions over binary trees

```
module BTFun where
import Prelude hiding (span, head, tail)
import Rank1
import Rank2
```

Binary trees without content.

```
data BTF a = Leaf | Span{left :: a, right :: a}
instance Functor BTF where
    fmap f Leaf = Leaf
    fmap f (Span t u) = Span (f t) (f u)
type BT = Mu1 BTF
leaf :: BT
leaf = In1 Leaf
span :: BT → BT → BT
span t u = In1 (Span t u)
```

Functions over binary trees as coinductive datatype. (Thorsten Altenkirch)

```
data TFunF f a = Cons{hd :: a, tl :: (f (f a))}
instance Functor2 TFunF where
    ffmap s f (Cons a t) = Cons (f a) (s (s f) t)
type TFun a = Nu2 TFunF a
```

Destructors.

```
head :: TFun a → a
head b = hd (out b)
tail :: TFun a → TFun (TFun a)
tail b = tl (out b)
```

Creating a TFun from a function over BT by coiteration.

```
newtype BTto a = BTto (BT → a)

stepLam :: BTto a → TFunF BTto a
stepLam (BTto f) = Cons (f leaf)
                        (BTto (λt → BTto (λu → f (span t u))))
lamBT′ :: (a → b) → (BT → a)    → TFun b
lamBT′ f g = Coit stepLam f (BTto g)
lamBT :: (BT → a) → TFun a
lamBT g = lamBT′ id g
```

Applying a TFun to a BT.

```
newtype TFto = TFto (forall a. TFun a → a)

stepApp :: BTF TFto → TFto
stepApp Leaf = TFto (λb → head b)
stepApp (Span (TFto l) (TFto r)) = TFto (λb → r (l (tail b)))

appBT :: BT → TFun a → a
appBT t = g where (TFto g) = it1 stepApp t
```

References

[AM03] Andreas Abel and Ralph Matthes. (Co-)iteration for higher-order nested datatypes. *TYPES'02 Proceedings*, 2003.