We will not use the missing simple quote mechanism for clarity.

1 HList

These examples are adapted from [Kiselyov(2004)]. Heterogeneous collections are used to handle symbol tables, XML elements, SQL rows, and others.

We don't need anymore the HList type class of Section 3 in [Kiselyov(2004)] since lifted lists are now kinded as lists whereas HList were kinded as star. We don't write functions as type classes with one element but as usual functions.

Here follows a few functions using HList. For each of them, we give its fully-explicit elaboration in FC.

```
data HList as where
  HNil :: HList '[]
  HCons :: a -> HList as -> HList (a ': as)
-- HList :: '[*] -> *
-- HNil  :: forall (r :: '[*]). ((~) '[*] r ('[] *)) -> HList r
-- HCons :: forall (r :: '[*]) (a :: *) (as :: '[*]).
--          ((~) '[*] r ((':) * a as)) -> a -> HList as -> HList r

hhead :: HList (a ': as) -> a
hhead (HCons x _xs) = x
-- head :: forall (a :: *) (as :: '[*]). HList (a ': as) -> a

htail :: HList (a ': as) -> HList as
htail (HCons _x xs) = xs
-- htail :: forall (a :: *) (as :: '[*]). HList (a ': as) -> HList as

hnull :: HList as -> Bool
hnull HNil = True
hnull (HCons _x _xs) = False
-- hnull :: forall (as :: '[*]). HList as -> Bool

hlength :: (Num n) => HList as -> n
hlength HNil = 0
hlength (HCons _x xs) = 1 + hlength xs
-- hlength :: forall (n :: *) (as :: '[*]). Num n -> HList as -> n

-- Might we need a proxy for 'p'?
hfoldr :: (forall (a :: *) (as :: '[*]). a -> p as -> p (a ': as))
       -> p '[] -> HList as -> p as
hfoldr _k z HNil = z
hfoldr k z (HCons x xs) = x `k` hfoldr k z xs
-- hfoldr :: forall (p :: '[*] -> *) (as :: '[*]).
--           (forall (a :: *) (as :: '[*]). a -> p as -> p (a ': as))
```

```
--           -> p '[] -> HList as -> p as
-- hfoldr = /\(p :: '[*] -> *) (as :: '[*])
--          \(k :: forall (a :: *) (as :: '[*]). a -> p as -> p (a ': as))
--          \(z :: p '[]) \(xs :: HList as)
--          case xs of
--            HNil c -> z |> sym c
--            HCons a as c x xs -> k a as x (hfoldr p as k z xs) |> sym c
```

Some functions defined in [Kiselyov(2004)] need computation in their types. For example, lookupAt which takes an HList and return its nth element will have a return type which depends on both the HList and the number. These functions are members of a type class using functional dependencies to compute the return type. These function starts with an H. We will separate the term function from its type function. The term function will be written as usual but with a function type application for its return type. The type function will be written with a type family and the lifted data constructors the term variant uses. Here are some examples for HAnd and HEq.

```
data Nat = Zero | Succ Nat
-- Nat :: *
-- Zero :: Nat
-- Succ :: Nat -> Nat

data HNat n where
  HZero :: HNat 'Zero
  HSucc :: HNat n -> HNat ('Succ n)
-- HNat  :: 'Nat -> *
-- HZero :: forall (r :: 'Nat). ((~) 'Nat r 'Zero) -> HNat r
-- HSucc :: forall (r :: 'Nat) (n :: 'Nat).
--          ((~) 'Nat r ('Succ n)) -> HNat n -> HNat r

hpred :: HNat ('Succ n) -> HNat n
hpred (HSucc n) = n
-- hpred :: forall (n :: 'Nat). HNat ('Succ n) -> HNat n

data HBool b where
  HFalse :: HBool 'False
  HTrue  :: HBool 'True
-- HBool  :: 'Bool -> *
-- HFalse :: forall (r :: 'Bool). ((~) 'Bool b 'False) -> HBool r
-- HTrue  :: forall (r :: 'Bool). ((~) 'Bool b 'True) -> HBool r

type family HAnd (b1 :: 'Bool) (b2 :: 'Bool) :: 'Bool
type instance HAnd 'False b = 'False
type instance HAnd 'True b = b
```

```
hand :: HBool b1 -> HBool b2 -> HBool (HAnd b1 b2)
hand HFalse _b = HFalse
hand HTrue b = b
-- hand :: forall (b1 :: 'Bool) (b2 :: 'Bool).
--         HBool b1 -> HBool b2 -> HBool (HAnd b1 b2)

type family HEq (m :: 'Nat) (n :: 'Nat) :: 'Bool
type instance HEq 'Zero 'Zero = 'True
type instance HEq ('Succ m) 'Zero = 'False
type instance HEq 'Zero ('Succ n) = 'False
type instance HEq ('Succ m) ('Succ n) = HEq m n

heq :: HNat m -> HNat n -> HBool (HEq m n)
heq HZero HZero = HTrue
heq (HSucc _m) HZero = HFalse
heq HZero (HSucc _n) = HFalse
heq (HSucc m) (HSucc n) = heq m n
-- heq :: forall (m :: 'Nat) (n :: 'Nat).
--        HNat m -> HNat n -> HBool (HEq m n)

type family HLength (as :: '[*]) :: 'Nat
type instance HLength '[] = 'Zero
type instance HLength (a ': as) = 'Succ (HLength as)

hlength' :: HList as -> HNat (HLength as)
hlength' HNil = HZero
hlength' (HCons _x xs) = HSucc (hlength' xs)
-- hlength' :: forall (as :: '[*]). HList as -> HNat (HLength as)

type family HLookupAt (n :: 'Nat) (as :: '[*]) :: *
type instance HLookupAt 'Zero ('Cons a as) = a
type instance HLookupAt ('Succ n) ('Cons a as) = HLookupAt n as

lookupAt :: HNat n -> HList as -> HLookupAt n as
lookupAt HZero (HCons x _xs) = x
lookupAt (HSucc n) (HCons _x xs) = lookupAt n xs
-- lookupAt :: forall (n :: 'Nat) (as :: '[*]).
--             HNat n -> HList as -> HLookupAt n as
```

Extensible records Let's take a look at labels to simulate extensible records. First let's define lifted strings for labels. In the following, prefix S stands for singleton.

```
data Char = CA|..|CZ|Ca|..|Cz
data SChar where
  SCA :: SChar 'CA
```

```
  ..
  SCz :: SChar 'Cz
-- SChar :: 'Char -> *
-- SCA   :: forall (r :: 'Char). (r ~ 'CA) -> SChar r

-- Ideally we could write the following
type family CharEq (c1 :: 'Char) (c2 :: 'Char) :: 'Bool
type instance CharEq c c = 'True
type instance CharEq _ _ = 'False

data SString where
  SNil  :: SString '[]
  SCons :: SChar c -> SString cs -> SString (c ': cs)
-- SString :: ['Char] -> *
-- SNil    :: forall (r :: ['Char]). (r ~ []) -> SString r
-- SCons   :: forall (r :: ['Char]) (c :: 'Char) (cs :: ['Char]).
--            (r ~ (c ': cs)) -> SChar c -> SString cs -> SString r

type lName = '['CN,'Ca,'Cm,'Ce]
sName :: SString lName
sName = SCons SCN $ SCons SCa $ SCons SCm $ SCons SCe SNil

type lAge = '['CA,'Cg,'Ce]
sAge :: SString lAge
sAge = SCons SCA $ SCons SCg $ SCons SCe SNil

type family StringEq (as :: ['Char]) (bs :: ['Char]) :: 'Bool
type instance StringEq '[] '[] = 'True
type instance StringEq (a ': as) '[] = 'False
type instance StringEq '[] (b ': bs) = 'False
type instance StringEq (a ': as) (b ': bs)
  = BoolAnd (CharEq a b) (StringEq as bs)

stringEq :: SString as -> SString bs -> HBool (StringEq as bs)
stringEq SNil SNil = STrue
stringEq (SCons a as) SNil = SFalse
stringEq SNil (SCons b bs) = SFalse
stringEq (SCons a as) (SCons b bs) = hand (charEq a b) (stringEq as bs)

type family ListMap (f :: (X -> Y)) (as :: '[X]) :: '[Y]
type instance ListMap f '[] = '[]
type instance ListMap f (a ': as) = f a ': ListMap f as

type family ListMember (eq :: X -> X -> 'Bool) (a :: X) (bs :: '[X]) :: 'Bool
type instance ListMember eq a '[] = 'False
type instance ListMember eq a (b ': bs) = BoolOr (eq a b) (ListMember a bs)
```

```
type family Fst (p :: '(X, Y)) :: X
type instance Fst '(a, b) = a

data Record where
  RNil  :: Record '[]
  RCons :: (ListMember StringEq l (ListMap Fst r) ~ 'False) =>
           SString l -> a -> Record r -> Record ('(l, a) ': r)
-- Record :: '['('['Char], *)] -> *
-- RNil   :: forall (rt :: '['('['Char], *)]). (rt ~ '[]) -> Record rt
-- RCons  :: forall (rt :: '['('['Char], *)])
--             (l :: '['Char]) (a :: *) (r :: '['('['Char], *)]).
--             (rt ~ ('(l, a) ': r))
--             -> (ListMember StringEq l (ListMap Fst r) ~ 'False)
--             -> SString l -> a -> Record r -> Record rt

rec :: Record '['(lName, String), '(lAge, Int)]
rec = RCons sName "Haskell" $ RCons sAge 21 RNil

type family If (b :: 'Bool) :: X -> X -> X
type instance If 'True t f = t
type instance If 'False t f = f

type family ListLookup (eq :: X -> X -> 'Bool) (a :: X) (r :: '['(X, Y)]) :: Y
type instance ListLookup eq a ('(b, c) ': r) = If (eq a b) c (ListLookup eq a r)

lookup :: SString l -> Record r -> ListLookup StringEq l r
lookup as (RCons bs x r) =
  case stringEq as bs of
    HTrue -> x
    HFalse -> lookup as r

-- TODO: Write update, delete, insert, proj.
-- TODO: It should be possible to simulate subtyping of records.
```

2 Generic table formatter

This example comes from [Chlipala(2010)].

Using extensible records as defined in previous section, we may want to write the following.

```
f :: Record '['(lName, String), '(lAge, Int)] -> String
f = mkTable
    $ RCons sName ("Name", id)
    $ RCons sAge ("Age", show)
```

```
      RNil

ghci> putStr $ f $ RCons sName "Haskell" $ RCons sAge 21 RNil
<tr> <th>Name</th> <td>Haskell</td> </tr>
<tr> <th>Age</th> <td>21</td> </tr>
```

Let's do it the same way it is done in Ur.

```
recfold :: (forall (l :: '['Char]) (a :: *) (r :: '['('['Char], *)]).
             SString l -> a -> p r -> p ('(l, a) ': r))
          -> p '[] -> Record r -> p r
recfold _k z RNil = z
recfold k z (RCons l x r) = k l x (recfold k z r)
-- recfold :: forall (p :: '['('['Char], *)] -> *) (r :: '['('['Char], *)]).
--            (forall (l :: '['Char]) (a :: *) (r :: '['('['Char], *)]).
--              SString l -> a -> p r -> p ('(l, a) ': r))
--            -> p '[] -> Record r -> p r

type family Meta (p :: '('['Char], *)) :: '('['Char], *)
type instance Meta '(l, t) = '(l, (String, t -> String))

-- without recfold
mkTable :: Record (ListMap Meta r) -> Record r -> String
mkTable RNil RNil = ""
mkTable (RCons l (th, td) mr) (RCons _l x r) =
  concat ["<tr> <th>",th,"</th> <td>",td x,"</td> </tr>"]
  ++ mkTable mr r

-- with recfold as in the paper
mkTable :: Record (ListMap Meta r) -> Record r -> String
mkTable = flip (recfold k z)
  -- [ p r ] is [ Record (ListMap Meta r) -> String ]
  where
    k :: SString l -> a -> (Record (ListMap Meta r) -> String)
         -> Record (ListMap Meta ('(l, a) ': r)) -> String
    k _l x f (RCons _l (th, td) r) =
      concat ["<tr> <th>",th,"</th> <td>",td x,"</td> </tr>"]
      ++ f r
    z :: Record (ListMap Meta '[]) -> String
    z _ = ""
```

3 Generic trie

This example comes from [Hinze et al.(2004)Hinze, Jeuring, and Lh].

All this is already possible in GHC right now, but with the well-kinding verification.

```
data Rep = TUnit | TChar | TSum Rep Rep | TProd Rep Rep
type family I (a :: 'Rep) :: *
type instance I 'TUnit = ()
type instance I 'TChar = Char
type instance I ('TSum a b) = Either (I a) (I b)
type instance I ('TProd a b) = (I a, I b)
data SRep where
  SUnit :: SRep 'TUnit
  SChar :: SRep 'TChar
  SSum  :: SRep a -> SRep b -> SRep ('TSum a b)
  SProd :: SRep a -> SRep b -> SRep ('TProd a b)
-- SRep :: 'Rep -> *

equal :: SRep a -> I a -> I a -> Bool
equal SUnit () () = True
equal SChar c1 c2 = equalChar c1 c2
equal (SSum a b) (Left x) (Left y) = equal a x y
equal (SSum a b) (Left x) (Right y) = False
equal (SSum a b) (Right x) (Left y) = False
equal (SSum a b) (Right x) (Right y) = equal b x y
equal (SProd a b) (x1, x2) (y1, y2) = equal a x1 y1 && equal b x2 y2

data Rep1 = T1Id | T1K Rep | T1Sum Rep1 Rep1 | T1Prod Rep1 Rep1
type family I1 (a :: 'Rep1) :: * -> *
type instance I1 'T1Id a = a
type instance I1 ('T1K t) _ = I t
type instance I1 ('T1Sum a b) c = Either (I1 a c) (I1 b c)
type instance I1 ('T1Prod a b) c = (I1 a c, I1 b c)
data SRep1 where
  S1Id :: SRep1 'T1Id
  S1K :: SRep a -> SRep1 ('T1K a)
  S1Sum  :: SRep1 a -> SRep1 b -> SRep1 ('T1Sum a b)
  S1Prod :: SRep1 a -> SRep1 b -> SRep1 ('T1Prod a b)
-- SRep1 :: 'Rep1 -> *

map :: SRep1 f -> (a -> b) -> I1 f a -> I1 f b
map S1Id m x = m x
map (S1K _) m c = c
map (S1Sum f g) m (Left x) = Left (map f m x)
map (S1Sum f g) m (Right x) = Right (map g m x)
map (S1Prod f g) m (x, y) = (map f m x, map g m y)

type family Trie (r :: 'Rep) :: * -> *
type instance Trie 'TUnit v = Maybe v
type instance Trie 'TChar v = TrieChar v
type instance Trie ('TSum a b) v = (Trie a v, Trie b v)
```

```
type instance Trie ('TProd a b) v = Trie a (Trie b v)
                     -- Nested type family application here

lookup :: SRep a -> I a -> Trie a b -> Maybe b
lookup SUnit () x = x
lookup SChar c f = lookupChar c f
lookup (SSum a b) (Left x) (t, _) = lookup a x t
lookup (SSum a b) (Right x) (_, t) = lookup b x t
lookup (SProd a b) (x, y) t = lookup a x t >>= lookup b y

empty :: SRep a -> Trie a b
  -- we might need a proxy for 'b' for type-inference of sum and prod
empty SUnit = Nothing
empty SChar = emptyChar
empty (SSum a b) = (empty a, empty b)
empty (SProd a b) = empty a
```

References

[Chlipala(2010)] A. Chlipala. Ur: Statically-typed metaprogramming with type-level record computation. In *PLDI'10*, pages 122–133, 2010.

[Hinze et al.(2004)Hinze, Jeuring, and Lh] R. Hinze, J. Jeuring, and A. Lh. Type-indexed data types. In *SCIENCE OF COMPUTER PROGRAMMING*, pages 148–174, 2004.

[Kiselyov(2004)] O. Kiselyov. Strongly typed heterogeneous collections. In *In Haskell 04: Proceedings of the ACM SIGPLAN workshop on Haskell*, pages 96–107. ACM Press, 2004.