INDUCTIVE TUPLES

Lennart Augustsson
Standard Chartered Bank

THE PROBLEM

- Every tuple is its own unique type, so

```
instance (Eq a1, Eq a2) => Eq (a1, a2) where
    (a1, a2) == (b1, b2) =
      a1 == b1 && a2 == b2

instance (Eq a1, Eq a2, Eq a3) => Eq (a1, a2, a3) where
    (a1, a2, a3) == (b1, b2, b3) =
      a1 == b1 && a2 == b2 && a3 == b3

instance (Eq a1, Eq a2, Eq a3, Eq a4) => Eq (a1, a2, a3, a4) where
    (a1, a2, a3, a4) == (b1, b2, b3, b4) =
      a1 == b1 && a2 == b2 && a3 == b3 && a4 == b4

instance (Eq a1, Eq a2, Eq a3, Eq a4, Eq a5) =>
         Eq (a1, a2, a3, a4, a5) where
    (a1, a2, a3, a4, a5) == (b1, b2, b3, b4, b5) =
      a1 == b1 && a2 == b2 && a3 == b3 && a4 == b4 && a5 == b5
```

THE PROBLEM

- One day I decided it was just too much

```
instance (HasType a1, HasType a2, HasType a3, HasType a4, HasType a5,
HasType a6, HasType a7, HasType a8, HasType a9, HasType a10, HasType
a11, HasType a12, HasType a13, HasType a14, HasType a15, HasType a16,
HasType a17, HasType a18, HasType a19, HasType a20, HasType a21, HasType
a22, HasType a23, HasType a24, HasType a25, HasType a26, HasType a27,
HasType a28, HasType a29, HasType a30, HasType a31, HasType a32, HasType
a33, HasType a34, HasType a35, HasType a36, HasType a37, HasType a38,
HasType a39, HasType a40, HasType a41, HasType a42, HasType a43, HasType
a44, HasType a45, HasType a46, HasType a47, HasType a48, HasType a49,
HasType a50, HasType a51, HasType a52) =>
        RelationRow (a1, a2, a3, a4, a5, a6, a7, a8, a9, a10, a11, a12,
a13, a14, a15, a16, a17, a18, a19, a20, a21, a22, a23, a24, a25, a26,
a27, a28, a29, a30, a31, a32, a33, a34, a35, a36, a37, a38, a39, a40,
a41, a42, a43, a44, a45, a46, a47, a48, a49, a50, a51, a52) where
    toRel = toRelTuple
    fromRow = fromITuple
```

THE SOLUTION

- The problem is that tuples don't have an inductive structure.
- Any solution to this problem will need some compiler support.
- We could use something like HList instead of tuples.
- I've opted for a slightly different solution.

THE SOLUTION

- A data type, indexed by a list of types

```
data Tuple (ts :: [*])
```

- So the regular tuples can be thought of as

```
type (a, b)       = Tuple '[a, b]
type (a, b, c)    = Tuple '[a, b, c]
type (a, b, c, d) = Tuple '[a, b, c, d]
…
```

- And even

```
type () = Tuple '[]
type OneTuple a = Tuple '[a]
```

- (From the onetuple package)

```
data OneTuple a = OneTuple a
```

THE SOLUTION

- The `Tuple` data type can be thought of as a GADT with an infinite number of constructors

```
data Tuple (ts :: [*]) where
    ()          ::                          Tuple '[]
    OneTuple a  :: a ->                     Tuple '[a]
    (,)         :: a -> b ->                Tuple '[a,b]
    (,,)        :: a -> b -> c ->           Tuple '[a,b,c]
    (,,,)       :: a -> b -> c -> d ->      Tuple '[a,b,c,d]
    …
```

- A data type with an infinite number of constructors needs compiler support.

- The equivalence between `Tuple` and regular tuples needs compiler support.

THE SOLUTION

- We also need some functions to operate on `Tuple` data.

```
class ATuple (as :: [*]) where
    consTuple   :: a -> Tuple as -> Tuple (a:as)
    unconsTuple :: Tuple (a:as) -> (a, Tuple as)
```

- We need an infinite number of instances, so it again needs compiler support.

SOME EXAMPLES

- Some examples:

```
instance Eq () where
    t1 == t2  =  True
instance (Eq a, Eq (Tuple as)) => Eq (Tuple (a : as)) where
    t1 == t2  =  a1 == a2 && as1 == as2
                 where (a1, as1) = unconsTuple t1
                       (a2, as2) = unconsTuple t2

instance Monoid () where
    mempty        = ()
    mappend _  _ = ()
instance (Monoid a, Monoid (Tuple as)) =>
         Monoid (Tuple (a : as)) where
    mempty         = consTuple mempty mempty
    mappend t1 t2 = consTuple (mappend a1 a2) (mappend as1 as2)
                    where (a1, as1) = unconsTuple t1
                          (a2, as2) = unconsTuple t2
```

CONCLUSIONS, PROS

- Finally allows unbounded instances for tuples.
- Moderate implementation effort. About two days in our compiler.
- With a few transformations in the optimiser it's as efficient as regular instances.
- Works well in practice.

CONCLUSIONS, CONS

- Requires compiler support.

- The GADT view of the tuple type is not totally satisfactory.
  What is the equivalent of the type constructor `(,)`?
  It would be something like `/\a./\b. Tuple [a, b]`, but that's not a Haskell type.