Dynamic Types in GHC 8

Stephanie Weirich
"A Reflection on Types"
joint work with Simon Peyton Jones,
Richard Eisenberg, and Dimitrios Vytiniotis

Compøse :: Conference 2016
Data.Dynamic

type Dynamic -- abstract

toDyn :: Typeable a => a -> Dynamic

fromDyn ::
  Typeable a => Dynamic -> Maybe a

  Marks presence of runtime type info

  Recover type via runtime check

dynlist :: [Dynamic]
dynlist = [toDyn "a", toDyn 2]
YOU COME TO ME WITH A COMPIL
TIME PROBLEM
AT RUNTIME?
memegenerator.net
Type systems should be a tool for programmers

Dynamic enforcement

Static enforcement
Dynamic types, why?

Suppose you wanted to implement something like the ST monad in GHC

type ST s a   -- state monad
type Ref s a   -- mutable reference

newRef :: Typeable a => a -> ST s (Ref s a)
readRef :: Typeable a => Ref s a -> ST s a
writeRef :: Typeable a =>
            Ref s a -> a -> ST s ()
runST :: (forall s. ST s a) -> a

Key ingredient: a data structure to store the values
Universal Store

data Ref s a = Ref Int

type S = Map Int Dynamic

extendStore :: Typeable a => Ref s a -> a -> S -> S
extendStore (Ref k) v s =
  Map.insert k (toDyn v) s

lookupStore :: Typeable a => Ref s a -> S -> Maybe a
lookupStore (Ref k) s = do
  d <- Map.lookup k s
  fromDyn d
How to implement Dynamic?

- **data Dynamic** = DInt Int | DBool Bool | DChar Char | DPair Dynamic Dynamic

- **toDynInt :: Int -> Dynamic**
  toDynInt = DInt

- **fromDynInt :: Dynamic -> Maybe Int**
  fromDynInt (DInt n) = Just n
  fromDynInt _ = Nothing

- **toDynPair :: Dynamic -> Dynamic -> Dynamic**
  toDynPair = DPair

- **dynFst :: Dynamic -> Maybe Dynamic**
  dynFst (DPair x1 x2) = Just x1
  dynFst _ = Nothing
Dynamics via Data.Typeable

class Typeable a where
    typeRep :: TypeRep a

trInt :: TypeRep Int
trInt = typeRep

Extensible:
Automatic instances from GHC

trIAI :: TypeRep (Int -> Int)
trIAI = typeRep

Indexed:
Type of representation tells you what it is
Dynamics via Data.Typeable

```haskell
class Typeable (a :: k) where
  typeRep :: TypeRep a

trInt :: TypeRep Int
trInt = typeRep

trIAI :: TypeRep (Int -> Int)
trIAI = typeRep

trArrow :: TypeRep (->)  
  Kind-polymorphic: Any kind of type
trArrow = typeRep
```
Dynamics via Type Reflection

data Dynamic where
  Dyn :: TypeRep a -> a -> Dynamic

toDyn :: Typeable a => a -> Dynamic
toDyn x = Dyn typeRep x

fromDyn :: forall a.
  Typeable a => Dynamic -> Maybe a
fromDyn (Dyn (rb :: TypeRep b) (x :: b)) =
  | ra == rb = Just x
  | otherwise = Nothing
  where
    ra = typeRep :: TypeRep a

Can't compare ra and rb with (==)
Just x has the wrong type
TypeRep Equality

-- Standard equality test
(==) :: Eq a => a -> a -> Bool

-- Equality test between type representations
eqT :: TypeRep a -> TypeRep b
    -> Maybe (a :~: b)
eqT = ... Arguments have different types
Returns an equality proof on success

-- Equality GADT, pattern matching shows a = b
data (a :~: b) where
    Refl :: a :~: a
TypeRep Equality

-- Equality test between type representations
eqT :: TypeRep a -> TypeRep b
    -> Maybe (a :~: b)

data (a :~: b) where
    Refl :: a :~: a

-- simple example using eqT
zero :: forall a. Typeable a => Maybe a
zero = do
    Refl <- eqT (typeRep :: TypeRep a)
          (typeRep :: TypeRep Int)
    return 0
Dynamics via Type Reflection

data Dynamic where  
    Dyn :: TypeRep a -> a -> Dynamic

toDouble :: Typeable a => a -> Dynamic
toDouble x = Dyn typeRep x

fromDouble :: forall a.  
    Typeable a => Dynamic -> Maybe a
fromDouble (Dyn rb x) = do  
    Refl <- eqT ra rb  
    return x  
    where ra = typeRep :: TypeRep a
Composing TypeReps

dynPair :: Dynamic -> Dynamic -> Dynamic
dynPair (Dyn r1 x1) (Dyn r2 x2) =
   Dyn (x1,x2)
Composing TypeReps (I)

dynPair :: Dynamic -> Dynamic -> Dynamic
dynPair (Dyn r1 x1) (Dyn r2 x2) =
    Dyn (trApp (trApp tPair r1) r2) (x1,x2)

tPair :: TypeRep (,)
tPair = typeRep

trApp :: TypeRep a -> TypeRep b -> TypeRep (a b)
trApp = ... primitive
Composing TypeReps (II)

dynPair :: Dynamic -> Dynamic -> Dynamic
dynPair (Dyn r1 (x1 :: a)) (Dyn r2 (x2 :: b)) =
  withTypeable r1 $ -- Typeable a
  withTypeable r2 $ -- Typeable b
  Dyn (typeRep :: TypeRep (a, b)) (x1,x2)

withTypeable :: TypeRep a ->
  (Typeable a => r) -> r
withTypeable = ... primitive

Explicitly provide type class evidence
Coherence: only GHC can create TypeReps
Decomposing Dynamics

dynFst :: Dynamic -> Maybe Dynamic
dynFst (Dyn rp x) = do
    Refl <- eqT rp
    (typeRep :: TypeRep (Dynamic,Dynamic))
    return (fst x)

example = do
    x <- dynFst (toDyn ('c', 'a'))
    y <- fromDyn x
    return $ y == 'c'

Returns False!
Decomposing TypeReps

How to determine the structure of a type representation?

splitApp :: TypeRep a -> Maybe (AppResult a)
splitApp = ... primitive

data AppResult (t :: k) where
  App :: TypeRep a1 -> TypeRep a2 --> AppResult (a1 a2)
Decomposing Dynamics with splitApp

dynFst :: Dynamic -> Maybe Dynamic
dynFst (Dyn rpab x) = do
  App rpa rb <- splitApp rpab
  -- know that x has type "pa b" here,
  -- for some types "pa" and "b"
  App rp ra <- splitApp rpa
  -- know that x has type "(p a) b" here
  Refl <- eqT rp (typeRep :: TypeRep (,))
  -- know that x has type (a,b) here
  return (Dyn ra (fst x))
Those are some fancy types…

- AppResult has an existential *kind*!

```haskell
data AppResult (t :: k) where
  App :: forall k1 (a :: k1 -> k) (b :: k1).
    TypeRep a -> TypeRep b -> AppResult (a b)
```

*Pattern match introduces a, b, and k1*
Decomposing Dynamics with splitApp

dynFst :: Dynamic -> Maybe Dynamic
dynFst (Dyn rpab x) = do
  App rpa rb <- splitApp rpab
  -- pa :: k1 -> *, b :: k1, pab = pa b
  App rp ra <- splitApp rpa
  -- p :: k2 -> k1 -> *, a :: k2, pa = p a
  Refl <- eqT rp (typeRep :: TypeRep (,))
  -- eqT must be polykinded and
  -- tell us that k1 == k2 == * & p == (,)
  return (Dyn ra (fst x))
Polykinded Equality

-- Equality test between type representations

eqT ::
    TypeRep a -> TypeRep b -> Maybe (a :~: b)

-- Equality proof type-indexed datatype

data a :~: b where
    Refl :: a :~: a
Polykinded Equality

-- Equality test between type representations
eqT :: forall k1 k2 (a :: k1) (b :: k2).
    TypeRep a -> TypeRep b -> Maybe (a :~: b)

-- Equality proof kind- and type- indexed
data (a :: k1) :~: (b :: k2) where
    Refl :: forall k (a :: k). a :~: a
-- Pattern matching tells us that a == b
-- AND k1 == k2
Summary

• Interface for safe runtime types
  • Type-indexed type representations provide safe usage of runtime types
  • withTypeable, singleton dictionary
  • Kind-polymorphic, heterogeneous equality

• Typeable enables extensible Dynamic type

• Available soon
  • Some parts (splitApp) require "Dependently-typed Haskell" features
Thanks!

Simon Peyton Jones,
Richard Eisenberg,
and Dimitrios Vytiniotis