RepLib: A library for derivable type classes

Stephanie Weirich

University of Pennsylvania

Generic Programming

- Aka *Type-Directed Programming*
- Behavior of *generic functions* determined by type information
  - Polymorphic equality, Read, Show
  - Reductions
  - Mapping
- Behavior determined by type structure and type name
  - Default case determined by structure
  - May be overridden by special case

Approaches to Generic Programming

- Domain-Specific Language
  - Generic operation specified in external language, compiled to Haskell
  - PolyP, Generic Haskell, Derivable type classes
- Generic functions
  - Define generic fold/map for each datatype
  - Define generic operation in terms of fold
  - Bimap, Scrap Your Boilerplate, Spines
- Type Reflection
  - Represent type structure with a datatype/GADT
  - Define operation with pattern matching & recursion
  - Typeable, First-class phantom types, RepLib

Representation types

- Singleton types that reflect type information in the term language
- Implemented in GHC with a GADT

```
data R a where
  Int    :: R Int
  Unit   :: R ()
  Pair   :: R a -> R b -> R (a,b)
  Arrow  :: R a -> R b -> R (a -> b)

Pair Int Unit :: R (Int, ())
```

Using Rep types

- Data constructor determines the type parameter

```
gsumR :: R a -> a -> Int
gsumR Int x = x
gsumR Unit x = 0
gsumR (Pair t1 t2) (x1,x2) =
        gsumR t1 x1 + gsumR t2 x2
gsumR (Arrow t1 t2) f = 0
```

Advantages of Rep types

- Definition by pattern matching/recursion
  - Not limited to simple folds
  - SYB-style combinators still available
  - Arrow types
- Dynamic types
  - Type-directed ops work even if type isn't known statically

```
data Dynamic = forall a. Dyn (a, R a)
```

Problems with Rep Types

- Using Rep types safely
- Using Rep types conveniently
- Representing datatypes generically
- Specializing generic operations

Using Rep types safely

- Not all operations are defined for all types

```
showR :: R a -> a -> String
showR (Arrow t1 t2) f = "<closure>"
```

- Other operations make even less sense for functions
- How to statically prevent showR from being called on Arrow type reps?

Using Rep types conveniently

- Often rep type argument is known statically

```
gsumR ((Char `Pair` Char) `Pair` Bool)
        ((‘a’,’b’), True)
```

- Constructing these reps by hand is tedious

Solution: type classes

- Use type class to provide representation

```
class Rep a where rep :: R a
instance Rep Int where rep = Int
instance (Rep a, Rep b) => Rep (a,b) where
  rep = Pair rep rep
class Rep a => GSum a where
  gsum :: a -> Int
  gsum = gsumR rep
```

- Instances declare what types are safe for each operation

```
instance GSum Int
instance (GSum a, GSum b) => GSum (a,b)
```

Generic datatype Reps

- All datatypes are *different* in Haskell

```
data Phone = Phone Int
data Age     = Age Int
f :: Age -> Age
f (Phone 1234567) ×
```

- Don't want new data constructor for each new datatype

```
data R a where
  Int       :: R Int
  Phone :: R Phone
  Age      :: R Age
  ...
```

- Want *datatype-generic* programming

Datatype-Generic Representation types

- Representation of a datatype constructor

```
data Con a =  ∀ l. Con (Emb l a) (RTup l)
```

- Heterogeneous list

```
data Nil = Nil
data a :*: l = a :*: l
```

- Embedding/Projection pair

```
data Emb l a = Emb { to      :: l -> a,
                     from :: a -> Maybe l }
```

- Rep of heterogeneous list

```
data RTup l where
  RNil  :: RTup Nil
  (:+:)   :: Rep a => R a -> RTup l -> RTup (a :*: l)
```

Generic view of datatypes

- Rep of data constructor

```
data Con a =  ∀ l. Con (Emb l a) (RTup l)
```

- Example

```
data Tree a = Leaf a | Node (Tree a) (Tree a)

leafEmb :: Emb (a :*: Nil) (Leaf a)
leafEmb = {
     to     = \(a :*: Nil) -> Leaf a,
     from = \x -> case x of Leaf a -> Just (a :*: Nil)
                                         _ -> Nothing
}
rLeaf :: forall a. Rep a => Con (Leaf a)
rLeaf = Con leafEmb ((rep::R a) :+: RNil)
```

Generic view of datatype

- List of data constructor reps

```
data R a where
    …
    Data :: DT -> [Con a] -> R a
```

- Plus name and reps of type args

```
data DT = ∀l.DT String (RTup l)
```

- Example

```
rTree :: forall a. Rep a => R (Tree a)
rTree = Data (DT "Tree" ((rep :: R a) :+: RNil))
                 [rLeaf,rNode]
```

Using generic rep

```
gsumR :: R a -> a -> Int
gsumR Int x = x
gsumR (Pair t1 t2) (x1,x2) = gsum t1 x1 + gsum t2 x2
gsumR (Data dt cons) x = findCon cons
  where
    findCon :: [Con a] -> Int
    findCon (Con emb reps : rest) =
      case (from emb x) of
          Just kids -> foldl_l (\r a b -> gsumR r a + b) 0 reps kids
          Nothing   -> findCon rest
    findCon [] = error “Impossible”
gsumR _ x = 0
```

Derivable type classes

- Provide instance of Rep class

```
instance Rep a => Rep (Tree a) where
    rep = rTree
```

- Derive instance of GSum class

```
instance GSum a => GSum (Tree a)
```

- Call generic function

```
gsum (Node (Leaf 4) (Leaf 5)) = 9
```

Specializing Generic Functions

```
newtype M = M Int
(derive [``M])

instance GSum M where
  gsum (M x) = 0

gsum (M 3) = 0
gsum (M 4, M 3) = 7
```

Solution

- Problem is in definition of gsumR for Pairs (and Data)
- If we could abstract over type classes…

```
data R c a where
  Int  :: R Int
  Pair :: (c a, c b) => R c (a,b)
  …

gsumR :: R GSum a -> a -> String
gsumR Int x = x
gsumR Pair (x1,x2) = gsum x1 + gsum x2
…

instance (c a, c b) => Rep c (a, b) where
  rep = Pair
instance Rep GSum a => GSum a where
  gsum = gsumR rep
```

What is RepLib?

- Definitions of Representation types
  - Both vanilla and parameterized reps
- Template Haskell automatically creates reps
- Generic functions (e.g. gsum)
- SYB-like combinators to define new ones (e.g. foldl_l, gmapT, gmapQ)
- Some support for type-*constructor* indexed functions (see paper)

Limitations

- Two different representation types
  - Parameterized representations not dynamic
  - Still other reps necessary for arity-2 and arity-3 generic functions (map and zip)
- No type-indexed types
  - Limited interaction between GADTs and MPTC
- No kind polymorphism/kind-indexed types
  - Args to type constructors must all be kind type
- Requires GHC extensions
- Can't represent all GHC types
- Dynamic type analysis

Conclusion

RepLib balances expressiveness and simplicity

RepLib can define

many common generic functions that analyze *many common* types in the *most popular* Haskell implementation.

Download now

- Help add to the library!
  - New generic functions
  - New combinators

- Available at:

  http://www.cis.upenn.edu/~sweirich/RepLib

Required GHC extensions

- Lexically-scoped type variables
- Higher-rank polymorphism
- Existential components
- GADTs
- Undecidable instances
- Template Haskell (for rep generation)