The Ins and Outs of Generic Programming

Hands-On Workshop @ Lambda World

Victor Cacciari Miraldo

Utrecht University

Preamble

Clone and Build:

```
.../ $ git clone
  https://github.com/VictorCMiraldo/lw2019-generics-workshop.git
.../ $ stack build
```

Motivation

Bob maintains networking code, Alice decides to add another field to some datatype used indirectly.

- No generics involved:
  - Compile-time failures if we have types
- Generics involved:
  - Nothing happens, generic infrastructure handles this automatically.

Well Known Generic Problems

- Equality
- Serialization
- Pretty-Printing
- Subterm Indexing
- Merkle Trees
- Differencing
- etc

Today

Three Generic Programming Libraries

- `GHC.Generics`, the builtin generics powerhorse
- `Generics.SOP`, with expressive combinator-based programming
- `Generics.MRSOP`, combinator-based programming with mutual recursion

The Core Idea

1. Represent datatypes with a uniform language
2. Interpret this language back into Haskell
3. Program over the uniform description

Datatype Building Blocks

Datatypes can be constructed with sums, products the unit type and the least fixpoint.

We can unwrap *one layer* of a recursive type:

```
data List a = Nil | Cons a (List a)

to :: Either () (a , List a) -> List a
to (Left ())        = Nil
to (Righ (x , xs)) = Cons x xs

from :: List a -> Either () (a , List a)
```

Or encode recursion explicitely:

```
newtype Fix f = Fix (f (Fix f))
newtype ListF a x = ListF (Either () (a , x))
```

Explicit Recursion

```
list123 :: Fix (ListF Int)
list123 = Fix (ListF (Right (1
        , Fix (ListF (Right (2
        , Fix (ListF (Right (3
        , Fix (ListF (Left ()))
        ))))))))
```

Pretty ugly...

Pattern synonyms make it evident these are the lists we know and love!

```
pattern Cons x xs  = Fix (ListF (Right x , xs))
pattern Nil        = Fix (ListF (Left ()))

list123 = Cons 1 (Cons 2 (Cons 3 Nil))
```

Datatype Building Blocks

Let's practice with a different type:

```haskell
data Bin a = Leaf | Fork a (Bin a) (Bin a)
```

Exercise 1:

```
.../lw2019-generics-workshop $ git checkout -b exercise-1
.../lw2019-generics-workshop $ emacs LW2019/Prelude.hs
```

Meet the first datatypes we will use today:

Discover:

```
.../lw2019-generics-workshop $ emacs LW2019/Types/Regular.hs
```

Datatype Building Blocks: Standardizing

`GHC.Generics` standard combinators instead of `Either`, `(,)`, …

```
data (f :*: g) x = f x :*: g x
data (f :+: g) x = L1 (f x) | R1 (g x)
data U1        x = U1
data K1 i c    x = K1 x
data M1 i c f  x = M1 x
data V1        x
```

Uniform language:

- Syntax: `data (f :+: g)`
- Intepretation: `L1 | R1`

Let’s write `GHC.Generic` representation of datatypes!

Exercise 2:

```
.../lw2019-generics-workshop $ git checkout -b exercise-2
.../lw2019-generics-workshop $ emacs LW2019/Generics/GHC/Repr.hs
```

Programming over Regular Datatypes

- Lists, Binary Trees, etc… Constructed using sums, products, unit and least fixpoints.

- `GHC.Generics` *does not* represent recursion explicitely.

- Standardized combinators allow us to write functions by *induction on the structure of the generic representation*.

```
instance (Func f , Func g) => Func (f :*: g) where
  func (fx :*: gx) = ...
```

Exercise 3:

```
.../lw2019-generics-workshop $ git checkout -b exercise-3
.../lw2019-generics-workshop $ emacs LW2019/Generics/GHC/Equality.hs
```

Sums-of-Products

- Induction on the typeclass level is long
- Haskell types already come in *normal form*! (SOP)

```haskell
data Bin a = Leaf a | Fork (Bin a) (Bin a)

type instance Code (Bin a) = '[ '[ a ]
                              , '[ Bin a , Bin a ] ]

type Rep (Bin a) = SOP I (Code (Bin a))
```

Uniform syntax is in `Code`. Interpretation is separate, with `SOP`.

Exercise 4:

```
.../lw2019-generics-workshop $ git checkout -b exercise-4
.../lw2019-generics-workshop $ emacs LW2019/Generics/SOP/Repr.hs
```

Sums-of-Products: Interpreting Codes (01)

Define GADT's that perform induction on *codes*:

```
data NS (f :: k -> *) :: [k] where
  Z :: f x     -> NS f (x ': xs)
  S :: NS f xs -> NS f (x ': xs)

data NP (f :: K -> *) :: [k] where
  Nil  :: NP f []
  Cons :: f x -> NP f xs -> NP f (x ': xs)
```

These are just n-ary sums and n-ary products. Think of it like:

```
NS f [x1 , x2 , ... , xn] == Either (f x1) (Either (f x2) ... (f xn))

NP f [x1 , x2 , ... , xm] == (f x1 , f x2 , ... , f xm)
```

Sums-of-Products: Interpreting Codes (02)

The whole recipe:

```
data NS (f :: k -> *) :: [k] where
  Z :: f x     -> NS f (x ': xs)
  S :: NS f xs -> NS f (x ': xs)

data NP (f :: K -> *) :: [k] where
  Nil  :: NP f []
  Cons :: f x -> NP f xs -> NP f (x ': xs)

newtype I x = I x

newtype SOP f code = SOP (NS (NP f) codes)
```

Lets write the equality function for sums of products

Exercise 5:

```
.../lw2019-generics-workshop $ git checkout -b exercise-5
.../lw2019-generics-workshop $ emacs LW2019/Generics/SOP/Equality.hs
```

Keep Note of These Types:

The `hcollapse` and `hczipWith` type signatures can hurt.

Here, we use them with the types:

```
newtype K a x = K a

hcollapse :: NP (K a) xs -> [a]

hczipWith :: (forall x . Eq x => f x -> g x -> h x)
          -> NP f xs -> NP g xs -> NP h xs
```

Is it possible to write a `hccollapse` version for `GHC.Generics`? Why?

No! `GHC.Generics` language encodes types in all shapes and forms. Imagine:

```
type T = f :*: (g :*: (h :*: (i :+: j)))
```

So Far

- Uniform Language to describe datatypes
- Interpret that back into Haskell
  - Implicitely, like `GHC.Generics`
  - Explicitely, like `Generics.SOP`
- Explicit interpretation has better programming support
  - Combinator-based approach versus typeclass

What's Missing?

- Recursion!

But why do we need it?

Sit tight...

Explicit Recursion Improvised

When do we start needing information about recursive structure?

```
shapeEq [1,2,3] [5,6,7] == True
shapeEq [1,2,3] [5,6]   == False
```

What changes from regular equality? Where we had,

```
class GEq a where ...
```

We now track the recursive the type,

```
class ShapeEq orig a where ...

class GShapeEq orig a where ...
  gshapeEq :: Proxy orig -> a -> a -> Bool
```

Recursion Example: `GHC.Generics`

Example Instance Search:

```
ShapeEq (Tree12 a)

GShapeEq (Tree12 a) (Rep (Tree12 a))

   GShapeEq (Tree12 a) (U1 :+: (K1 R a :*: K1 R (Tree12 a)) :+: ...)

      GShapeEq (Tree12 a) U1 -- Ok!

      GShapeEq (Tree12 a) (K1 R a :*: K1 R (Tree12 a))

         GShapeEq (Tree12 a) (K1 R a)

         GShapeEq (Tree12 a) (K1 R (Tree12 a))
```

Recursion Example: `GHC.Generics`

Use `OVERLAPPING` instances!

```
instance {-# OVERLAPPING  #-} (ShapeEq orig orig)
  => GShapeEq orig (K1 orig) where ...

instance {-# OVERLAPPABLE #-} GShapeEq orig (K1 a) where ...
```

Exercise 6:

```
.../lw2019-generics-workshop $ git checkout -b exercise-6
.../lw2019-generics-workshop $ emacs LW2019/Generics/GHC/ShapeEquality.hs
```

And yes... We have to use the `orig` trick every time we need to have information about which fields of a constructor are recursive occurences of our type.

Explicit Recursion Rehearsed: `SOP`

The `generics-sop` approach saves some work. We only need to do the `orig` work once:

Define an annotated type.

```
data Ann orig :: * -> * where
  Rec   :: orig -> Ann orig orig
  NoRec :: x    -> Ann orig x
```

And define instances to annotate n-ary products.

```
class AnnotateRec orig (prod :: [ * ]) where
  annotate :: NP I prod -> NP (Ann orig) prod
```

Discover:

```
.../lw2019-generics-workshop $ emacs LW2019/Generics/SOP/AnnotateRec.hs
```

Explicit Recursion Rehearsed: `SOP`

Let's define shape equality for `SOP` and compare!

Exercise 7:

```
.../lw2019-generics-workshop $ git checkout -b exercise-7
.../lw2019-generics-workshop $ emacs LW2019/Generics/SOP/ShapeEquality.hs
```

- How far is `Generics.GHC.Equality` to `Generics.GHC.ShapeEquality`?
- How far is `Generics.SOP.Equality` from `ShapeEquality`?

That's a consequence of the *combinator based*, which is only possible because the interpretation of the generic language is *explicitly* for types in a normal form.

Explicit Recursion Composed

The `generics-mrsop` library supports Mutually Recursive Types, which are a superset of the regular types.

- Regular:

```
data [a]     = [] | a : [a]
data Tree a  = Leaf | Bin a (Tree a) (Tree a)
data Maybe a = Nothing | Just a
```

- Mutually Recursive:

```
data Zig = Zig | ZigZag Zag
data Zag = Zag | ZagZig Zig
```

Codes get lifted from to `[ [ [Atom kon] ] ]`, where

```
data Atom kon = K kon | I Nat
```

Codes for Mutually Recursive Types

```
data Zig = Zig | ZigZag Zag
data Zag = Zag | ZagZig Zig

type FamZig  = '[Zig , Zag]

type CodeZig = '[ '[ '[] , '[ I 1 ] ]
               , '[ '[] , '[ I 0 ] ] ]

type GZig = Rep Opaques (Lkup FamZig) (Lkup CodesZig 0)

type family Lkup [x1 , ... xn] m = xm
```

The main difference from SOP is the NA:

```
newtype Rep ki f code = Rep (NS (NP (NA ki f)) code

data NA ki f :: Atom -> * where
  NA_I :: f ix -> NA (I ix)
  NA_K :: ki k -> NA (K k)
```

The Sugar-free `Generic` Class

```
class Family (ki :: kon -> *) (fam :: [*]) (codes :: [[[Atom kon]]])
  where
    sfrom' :: SNat ix -> El fam ix -> Rep ki (El fam) (Lkup ix codes)

    sto'   :: SNat ix -> Rep ki (El fam) (Lkup ix codes) -> El fam ix

data SNat :: Nat -> * where
  SZ :: SNat Z
  SS :: SNat n -> SNat (S n)

data El :: [k] -> Nat -> k where
  El :: Lkup fam ix -> El fam ix
```

Exercise 8:

```
.../lw2019-generics-workshop $ git checkout -b exercise-8
.../lw2019-generics-workshop $ emacs LW2019/Generic/MRSOP/Repr.hs
```

Equalities in `generics-mrsop`

Exercise 9:

```
.../lw2019-generics-workshop $ git checkout -b exercise-9
.../lw2019-generics-workshop $ emacs LW2019/Generic/MRSOP/Equality.hs
```

Exercise 10:

```
.../lw2019-generics-workshop $ git checkout -b exercise-10
.../lw2019-generics-workshop $ emacs LW2019/Generic/MRSOP/ShapeEquality.hs
```

```haskell
zipRep :: Rep ki f c -> Rep kj g c
       -> Maybe (Rep (ki :*: kj) (f :*: g) c)

elimRep :: (forall k.  ki k  -> a)  -- eliminate opaques
        -> (forall ix. f  ix -> a)  -- eliminate recursive positions
        -> ([a] -> b)               -- combine the eliminated fields
        -> Rep ki f c               -- value we want to eliminate
        -> b
```

Catamorphisms (AKA `fold`)

Explicit Recursion enables generic recursion schemes!

```
cata :: (forall iy. Rep ki phi (Lkup iy codes) -> phi iy)
     -> Fix ki codes ix
     -> phi ix
```

Exercise 11:

```
.../lw2019-generics-workshop $ git checkout -b exercise-11
.../lw2019-generics-workshop $ emacs LW2019/Generic/MRSOP/Height.hs
```

Annotated Fixpoints

Instead of consuming a type, we can choose to keep the intermediary results annotated in the tree.

```
newtype AnnFix phi f = AnnFix (phi , f (AnnFix phi f))
```

Annotated Fixpoints: `generics-mrsop`

In `mrsop`, we need a slightly more complicated type, because of the indicies involved.

```
synthesize :: (forall iy . Rep ki phi (Lkup iy codes) -> phi iy)
           -> Fix ki codes ix
           -> AnnFix ki codes phi ix
```

where

```
newtype AnnFix ki codes (phi :: Nat -> *) ix = ...
```

Advanced Material

If you are into this kind of things, make sure to check the rest of the repository. For example,

Discover:

```
.../lw2019-generics-workshop $ emacs LW2019/Generics/MRSOP/Arbitrary.hs
```

Summary

	Pattern Functors	Codes
No Explicit Recursion	`GHC.Generics`[2]	`generics-sop`[1]
Simple Recursion	`regular`[6]	`generics-mrsop`[3]
Mutual Recursion	`multirec`[7]	

Other approaches include support for GADT's[4] and higher kinded classes[5].

Reading Material i

[1] E. de Vries and A. Löh. True sums of products. In *Proceedings of the 10th ACM SIGPLAN Workshop on Generic Programming*, WGP '14, pages 83–94, New York, NY, USA, 2014. ACM.

[2] J. P. Magalhães, A. Dijkstra, J. Jeuring, and A. Löh. A generic deriving mechanism for haskell. In *Proceedings of the Third ACM Haskell Symposium on Haskell*, Haskell '10, pages 37–48, New York, NY, USA, 2010. ACM.

[3] V. C. Miraldo and A. Serrano. Sums of products for mutually recursive datatypes: The appropriationist's view on generic programming. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Type-Driven Development*, TyDe 2018, pages 65–77, New York, NY, USA, 2018. ACM.

[4] A. Serrano and V. C. Miraldo. Generic programming of all kinds. In *Proceedings of the 11th ACM SIGPLAN International Symposium on Haskell*, Haskell 2018, pages 41–54, New York, NY, USA, 2018. ACM.

[5] A. Serrano and V. C. Miraldo. Classes of arbitrary kind. In J. J. Alferes and M. Johansson, editors, *Practical Aspects of Declarative Languages*, pages 150–168, Cham, 2019. Springer International Publishing.

[6] T. VAN NOORT, A. RODRIGUEZ YAKUSHEV, S. HOLDERMANS, J. JEURING, B. HEEREN, and J. P. MAGALHÃES. A lightweight approach to datatype-generic rewriting. *Journal of Functional Programming*, 20(3-4):375–413, 2010.

[7] A. R. Yakushev, S. Holdermans, A. Löh, and J. Jeuring. Generic programming with fixed points for mutually recursive datatypes. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming*, ICFP '09, pages 233–244, New York, NY, USA, 2009. ACM.

Utrecht University

The Ins and Outs of Generic Programming

Hands-On Workshop @ Lambda World

Victor Cacciari Miraldo

Utrecht University