Generic programming for the masses

Ryan Scott

Indiana University

github.com/RyanGlScott

rgscott@indiana.edu

April 12, 2017

Common scenario

- We have a function we want to implement for many data types
- Equality, comparison, pretty-printing, etc.
- In Haskell, usually accomplished via type classes

One example

Another example

Notice a pattern?

- All `Equal` instances have the very similar structure
- Tedious, and requires maintenance
- Surely there must be a way to automate this!

Common ways to automate the creation of instances

- Built-in language support (straight-up magic)
- Macros (slightly more palatable magic)

Built-in language support

- You can automatically derive certain privileged typeclasses
- The `deriving Show` bit causes this code to be generated:

`deriving` drawbacks

- Complete hides the algorithm from users (unless you want to dig through GHC's source to understand it!)
- If your type class isn't one of the exalted few (`Eq`, `Ord`, `Read`, `Show`, etc.), you can't use `deriving` with it

Macros

- GHC has macro system called Template Haskell (TH)
- Allows programmers to
  - Reify info about top-level definition
  - Quasiquote Haskell source code into a manipulatable TH AST
  - Splice TH AST back into source code
- Can be leveraged to derive instances

Template Haskell example

(Code by Sami Hangaslammi: `https://gist.github.com/shangaslammi/1524967`)

Template Haskell example (cont'd.)

- Typing this into Haskell source code:
- Splices this instance (visible via `ghc -ddump-splices`):

Template Haskell drawbacks

- TH is a nice tool when you can use it, but...
  - Learning curve
  - Staging issues
  - Ugly as sin
  - Still requires several gallons of magic to work

We need a better way to automatically derive instances

- Goals:
  1. Make the instance deriving algorithm transparent to programmers
  2. Extensible to many type classes
  3. Use as much pure Haskell as possible (minimize magic)

Aside: regular datatypes form an algebra!

- We can form an algebra (semiring) out of Haskell datatypes (Yorgey and Piponi)
- A *semiring* is a set R with:
  - Associative operations $+$, $\bullet$ with identities 0, 1
  - $+$ is commutative
  - $\bullet$ distributes over $+$
  - Neither $+$ nor $\bullet$ necessarily have inverses
- Examples: $(\mathbb{N}, +, \times)$, $(\{\mathtt{true}, \mathtt{false}\}, \mathtt{or}, \mathtt{and})$

Putting the algebra in algebraic datatypes

- We can view Haskell datatypes abstractly through the lens of *polynomial functors*
- We inductively define the universe **Fun** of polynomial functors:
  - Constant functors: $K_A \in \mathbf{Fun}$ where $K_A\ a = A$
  - Identity functor: $X \in \mathbf{Fun}$ where $X\ a = a$
  - Sums of functors: $\forall F, G \in \mathbf{Fun},\ F + G \in \mathbf{Fun}$ where $(F + G)\ a = F\ a + G\ a$
  - Products of functors: $\forall F, G \in \mathbf{Fun},\ F \times G \in \mathbf{Fun}$ where $(F \times G)\ a = F\ a \times G\ a$
    - Abbreviate $F \times G$ as FG

Putting the algebra in algebraic datatypes (cont'd)

- Polynomial functors form a semiring under $+$ and $\times$, where $1 = K_{\texttt{Unit}}$ and $0 = K_{\texttt{Void}}$
- Haskell datatypes are isomorphic to polynomial functors!
- Examples:
  - `data Bool = False | True`
    $B = 1 + 1$
  - `data List a = Nil | Cons a (List a)`
    $L(A) = 1 + A \times L(A)$
  - `data Tree a = Leaf | Node a | Branch (Tree a) a (Tree a)`
    $T(A) = 1 + A + A \times T(A)^2$

Putting it into code

- Translating this encoding of datatypes to Haskell proves straightforward:

Putting it into code

- Continuing the previous examples:
  -
  -
  -
- Now we have a common vocabulary for talking about any datatype!

Implementing `Equal` generically

- Recall our earlier example:
- Using our new generic datatype technology, we should be able to derive `Equal` instances with ease
- To that end, let's invent a generic `Equal` counterpart:[1]

[1] Note that the parameter in `Equal` is of kind `*`, but the one in `GEqual` is of kind `* -> *`. More on this later.

Implementing `Equal` generically

- Case 1: `data U1 p = U1`
  - A nullary constructor is always equal to itself

Implementing `Equal` generically

- Case 2: `data (f :+:  g) p = L1 (f p) | R1 (g p)`
  - One branch of a sum is only equal to the another value from the same branch (and only if the underlying types are equal)

Implementing `Equal` generically

- Case 3: `data (f :*:  g) p = f p :*:  g p`
  - A product is equal to another product if its constituent types are equal to the corresponding types in the other pair

Implementing `Equal` generically

- Case 4: `newtype Rec0 c p = Rec0 c`
  - For constants, defer to the underlying `Equal` instance:

Implementing `Equal` generically

- Case 5: `data V1 p`
  - If a datatype is not inhabited by any values, we punt.

Implementing `Equal` generically

- Now we need a way to use `GEqual` in an `Equal` instance
- Solution: another typeclass!
- Example instance:

Implementing `Equal` generically

- Now implementing an `Equal` instance for any `Generic` instance is a breeze!

Further elimination of boilerplate

- We inadvertently introduced more boilerplate by having to define `Generic` instances
- To remedy this, we'll introduce one small piece of magic. This:
- can be done with this:

Further elimination of boilerplate

- There's also the `eq = genericEq` boilerplate.
- Use default instance signatures to get around this:
- Now you don't have to implement the default definition yourself:

Further elimination of boilerplate

- You can get the best of both worlds with GHC 7.10's `-XDeriveAnyClass` extension:

What else can you do with generics?

- You can encode metadata with another representation type:

Caveats

- GHC generics can incur a runtime cost due to conversion to/from representation types
  - Good chance representation types can be inlined away, though
- Cannot handle certain sophisticated type features, e.g.,

Takeaways

- A generic programming technique with a much lower learning curve
- Eliminates large swaths of boilerplate
- Avoids many of the frustrations of `deriving` and Template Haskell

Any questions?

How GHC generics gets its metadata (pre-GHC 8.0)

- `-XDeriveGeneric` generates proxy datatypes for metadata instances:

How GHC generics gets its metadata (GHC 8.0 and later)

- Encode the metadata in the type!
- Uses singleton types reify the type information as a value:
- No need to generate any extra datatypes or instances!

Generic1

- There's also a way to generically implement typeclasses of kind `* -> *`:
- An example of a typeclass of kind `* -> *`:

Generic1

- We can generically derive `Mappable` using the same machinery!