Fun and **generic** things to do with **EMGM**

Sean Leather

9 July 2009

Extensible and Modular Generics for the Masses

EMGM is a powerful library that uses type classes for **datatype-generic programming (DGP)** in Haskell.

The **`emgm`** package on Hackage provides the following:

- Documented platform for writing generic functions
- Flexible functionality for deriving instances using Template Haskell
- Growing collection of useful generic functions

History of EMGM

1. Published as **Generics for the Masses** by Ralf Hinze in 2004.
2. Revised by Bruno Oliveira, Andres Löh, and Hinze for extensibility and modularity in 2006.
3. Explored further and compared with other DGP libraries by Alexey Rodriguez Yakushev et al in 2007-2008.
4. Packaged and released by Sean Leather, José Pedro Magalhães, and others at Utrecht University in September 2008.

A tutorial is available as part of lecture notes created for the 2008 Advanced Functional Programming Summer School.

`http://www.cs.uu.nl/research/techreps/UU-CS-2008-025.html`

Overview

- Datatype-Generic Programming
- Representing Datatypes in EMGM
- Defining, Using, Extending Generic Functions
  - Empty
  - Crush
  - Ad hoc Instances
  - Collect
- Continuing Development of EMGM

Datatype-Generic Programming

- The term was coined by Jeremy Gibbons several years ago, but the technique has been around for around a decade.
- Scrap Your Boilerplate (SYB) is an example of a popular DGP library.
- DGP means generic on the **structure of a datatype**.

Structure of a Datatype

- The structure is a way of representing the common aspects of many datatypes, e.g. constructors, alternatives, tupling.
- An intuitive way to determine the structure of a datatype is to look at its declaration.

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

- There are multiple **generic views** of the structure.
- SYB uses one based on combinators.
- EMGM uses a different one based on binary sums of products.

Representing Structure in EMGM (1)

$$\textbf{data}\ \mathsf{Tree}\ \mathsf{a} = \mathsf{Tip} \mid \mathsf{Leaf}\ \mathsf{a} \mid \mathsf{Node}\ \mathsf{Int}\ (\mathsf{Tree}\ \mathsf{a})\ (\mathsf{Tree}\ \mathsf{a})$$

To view the Tree type in its structure representation, we can substitute its syntax with (nested) sums (alternatives) and products (pairs).

$$\textbf{type}\ \mathsf{Tree}^{\circ}\ \mathsf{a} = \mathbb{1} + \mathsf{a} + \mathsf{Int} * \mathsf{Tree}\ \mathsf{a} * \mathsf{Tree}\ \mathsf{a}$$

Another way of defining Tree° is using standard Haskell types.

$$\textbf{type}\ \mathsf{Tree}^{\circ}\ \mathsf{a} = \mathsf{Either}\ ()\ (\mathsf{Either}\ \mathsf{a}\ (\mathsf{Int}, (\mathsf{Tree}\ \mathsf{a}, \mathsf{Tree}\ \mathsf{a})))$$

Representing Structure in EMGM (2)

While we might use standard Haskell types, we choose to use our own types for better readability and to prevent confusion between datatypes used in the representation and those that are represented.

```
data Unit    = Unit        -- ()
data a :*: b = a :*: b     -- (a, b)
data a :+: b = L a | R b   -- Either a b
```

This is the structure we use for EMGM.

```
type Tree° a = Unit :+: a :+: Int :*: Tree a :*: Tree a
```

We will also need descriptions of the constructors and types.

```
data ConDescr  = ConDescr ...
data TypeDescr = TypeDescr ...
```

Representing Structure in EMGM (3)

In order to access the structure of a datatype, we need to translate a value from its native form to a representation form.

This is done using an **isomorphism** implemented as an **embedding-projection pair**.

```
data EP d r = EP {from :: (d → r), to :: (r → d)}
```

Here is the EP for Tree.

```
epTree :: EP (Tree a) (Tree° a)
epTree = EP fromTree toTree
  where fromTree Tip                = L Unit
        fromTree (Leaf a)           = R (L a)
        fromTree (Node i t1 t2)     = R (R (i :*: t1 :*: t2))

        toTree (L Unit)                    = Tip
        toTree (R (L a))                   = Leaf a
        toTree (R (R (i :*: t1 :*: t2)))   = Node i t1 t2
```

Representing Structure in EMGM (4)

A generic function is written by induction on the structure of a datatype. We represent the cases of a function as methods of this (summarized) type class Generic.

```
class Generic g where
  rint   :: g Int
   ...
  runit  :: g Unit
  rsum   :: g a → g b → g (a :+: b)
  rprod  :: g a → g b → g (a :*: b)
  rcon   :: ConDescr → g a → g a
  rtype  :: TypeDescr → EP b a → g a → g b
```

Our **universe** supports constant types, structure types, and the ability to extend the universe with arbitrary datatypes using rtype.

Representing Structure in EMGM (5)

To add a new datatype representation, we need to define an rtype value.

```
rTree :: (Generic g, Rep g a, Rep g Int, Rep g (Tree a)) => g (Tree a)
rTree = rtype (TypeDescr ... ) epTree
                (rcon (ConDescr ... ) runit `rsum`
                 rcon (ConDescr ... ) rep   `rsum`
                 rcon (ConDescr ... ) (rep `rprod` rep `rprod` rep))
```

But what is this Rep and rep about?

Representing Structure in EMGM (6)

To avoid having to provide all of these representations for every function, we use another type class, Rep.

```
class Rep g a where
  rep :: g a
instance (Generic g, Rep g a, Rep g Int, Rep g (Tree a)) ⇒
           Rep g (Tree a) where
  rep = rTree
```

Of course, we also need instances for the constant types.

```
instance (Generic g) ⇒ Rep g Int where
  rep = rint
 ...
```

Generating the Structure Representation

Fortunately, we don't have to write all of the previous boilerplate. We can generate it using the Template Haskell functions included in the `emgm` package.

```
$(derive ''Tree)
```

This creates the EP, the ConDescr and TypeDescr, and all class instances needed.

It is a good idea to understand what code is being derived. Use the following pragma or command-line option in GHC to see the code generated at compile time:

```
{-# OPTIONS -ddump-splices #-}
```

First Generic Function: Defining Empty (1)

Now, we're ready to write our first generic function. Recall the Generic class (in full).

```
class Generic g where
  rconstant :: (Enum a, Eq a, Ord a, Read a, Show a) ⇒ g a
  rint      :: g Int
  rinteger  :: g Integer
  rfloat    :: g Float
  rdouble   :: g Double
  rchar     :: g Char
  runit     :: g Unit
  rsum      :: g a → g b → g (a :+: b)
  rprod     :: g a → g b → g (a :*: b)
  rcon      :: ConDescr → g a → g a
  rtype     :: TypeDescr → EP b a → g a → g b
```

A generic function is an instance of Generic. To write a function, we need to produce a type for the instance.

Defining Empty (2)

The simple function we're going to write is called Empty. It returns the value of a type that is traditionally the initial value if you were to enumerate all values. (Enum is included in `emgm`.)

The type of the function is enclosed in a **newtype**.

```haskell
newtype Empty a = Empty {selEmpty :: a}
```

Note that the type of selEmpty gives a strong indication of the type of the final function. For Empty, the type is identical (modulo class constraints), but for some functions, it can change.

Defining Empty (3)

The function definition is straightforward.

```
instance Generic Empty where
  rconstant          = error "Should not be called!"
  rint               = Empty 0
  rinteger           = Empty 0
  rfloat             = Empty 0
  rdouble            = Empty 0
  rchar              = Empty '\NUL'
  runit              = Empty Unit
  rsum         ra rb = Empty (L (selEmpty ra))
  rprod        ra rb = Empty (selEmpty ra :*: selEmpty rb)
  rcon  cd     ra    = Empty (selEmpty ra)
  rtype td ep  ra    = Empty (to ep (selEmpty ra))
```

Defining Empty (4)

The "core" generic function is $\mathsf{selEmpty} :: \mathsf{Empty\ a} \rightarrow \mathsf{a}$, but we wrap it with a more usable function:

$$
\begin{aligned}
&\mathsf{empty} :: (\mathsf{Rep\ Empty\ a}) \Rightarrow \mathsf{a} \\
&\mathsf{empty} = \mathsf{selEmpty\ rep}
\end{aligned}
$$

The primary purpose of Rep is to dispatch the appropriate type representation.

Applying empty:

$$\mathsf{test1} = (\mathsf{empty} :: \mathsf{Tree\ Int}) == \mathsf{Tip}$$

Empty has a very simple definition, and it may not be extremely useful, but it demonstrates the basics of defining a generic function.

Let's move on to a more complicated function that is also much more useful.

In Over Our Heads: Defining Crush (1)

- The generic function Crush is sometimes called a generalization of the list “fold” operations — but so is a catamorphism.
- It is also sometimes called “reduce” — not exactly a precise description.
- To avoid confusion, let’s not do any of these things and just focus on how it works.

Defining Crush (2)

- Crush operates on the elements of a container or functor type.
- It traverses all of the elements and accumulates a result that combines them in some way.
- In order to do this, Crush requires a nullary value to initialize the accumulator and a binary operation to combine an element with the accumulator.

We will define a function with a type signature similar to this:

$$\mathsf{crush} :: (\ldots) \Rightarrow (\mathsf{a} \to \mathsf{b} \to \mathsf{b}) \to \mathsf{b} \to \mathsf{f}\ \mathsf{a} \to \mathsf{b}$$

Notice the similarity:

$$\mathsf{foldr} :: (\mathsf{a} \to \mathsf{b} \to \mathsf{b}) \to \mathsf{b} \to [\mathsf{a}] \to \mathsf{b}$$

Defining Crush (3)

$$\mathsf{crush} :: (\ldots) \Rightarrow (\mathsf{a} \to \mathsf{b} \to \mathsf{b}) \to \mathsf{b} \to \mathsf{f\ a} \to \mathsf{b}$$

Our first challenge is to define the **newtype** for the function. Recall that this gives a strong indication of the type of the function, but that it doesn't necessarily match the final type exactly. Let's try to determine that type.

We have several major differences between the requirements for Empty and those for Crush.

1. Crush takes arguments.
2. Crush has three type variables over Empty's one.
3. Crush deals with a functor type (i.e. $\mathsf{f} :: \star \to \star$).

Let's see how to deal with these.

Defining Crush (4)

$$\mathsf{crush} :: (\ldots) \Rightarrow (\mathsf{a} \to \mathsf{b} \to \mathsf{b}) \to \mathsf{b} \to \mathsf{f}\ \mathsf{a} \to \mathsf{b}$$

1. **Crush takes arguments.** This is not difficult to handle. Our generic function cases can take arguments, too.

2. **Crush has three type variables over Empty's one.** When defining a generic function in EMGM, it is important to determine which types are actually "generic" (i.e. will need a structure representation) and which types are not (e.g. may be polymorphic).

   In this case, we traverse only the structure of the container, so the only truly generic type variable is f. Variables a and b are polymorphic.

2. **Crush deals with a functor type (i.e. $\mathsf{f} :: \star \to \star$).** Unfortunately, our current representation does not handle this. Fortunately, the change is not large.

Defining Crush (5)

We need a type class representation dispatcher for functor types.

```
class FRep g f where
  frep :: g a → g (f a)
```

FRep allows us to represent the structure of a functor type while also giving us access to the element type contained within.

Reusing our Tree example:

```
instance (Generic g) ⇒ FRep g Tree where
  frep ra =
    rtype (TypeDescr ...) epTree
          (rcon (ConDescr ...) runit `rsum`
           rcon (ConDescr ...) ra    `rsum`
           rcon (ConDescr ...) (rint `rprod` frep ra `rprod` frep ra))
```

Again, this is generated code, and we don't have to write it.

Defining Crush (6)

Now, back to defining the **newtype** for our function.

$$\mathsf{crush} :: (\ldots) \Rightarrow (\mathsf{a} \to \mathsf{b} \to \mathsf{b}) \to \mathsf{b} \to \mathsf{f}\ \mathsf{a} \to \mathsf{b}$$

- We need to determine the core functionality and choose the most general type.
- The core generic functionality is to combine a value from a structure representation with a non-generic value and return a non-generic result. This is effectively the higher-order argument.

$$\textbf{newtype}\ \mathsf{Crush}\ \mathsf{b}\ \mathsf{a} = \mathsf{Crush}\ \{\mathsf{selCrush} :: \mathsf{a} \to \mathsf{b} \to \mathsf{b}\}$$

We must be careful, however, to avoid thinking that this is the exact same as the combining function. We are indicating two important aspects with this declaration:

1. Type of the "core" generic function: type of selCrush
2. Which types are generic: type parameters of Crush

Defining Crush (7)

Next, we define the function cases themselves.

```
instance Generic (Crush b) where
```

The constant types (including Unit) are simple. The constructor case is almost as simple. The rtype case adds the conversion from the datatype.

```
rconstant        = Crush (const id)
rcon  cd         = Crush · selCrush
rtype td ep ra   = Crush (selCrush ra · from ep)
```

The sum case is somewhat more interesting.

```
rsum ra rb = Crush go
  where go (L a) = selCrush ra a
        go (R b) = selCrush rb b
```

Defining Crush (8)

The product case is even more interesting.

```
rprod ra rb = Crush go
  where go (a :*: b) = selCrush ra a · selCrush rb b
```

Or should it be...?

```
rprod ra rb = Crush go
  where go (a :*: b) = selCrush rb b · selCrush ra a
```

Defining Crush (9)

Fortunately, we can turn this problem into a choice.

```
data Assoc = AssocLeft
           | AssocRight
newtype Crush b a = Crush {selCrush :: Assoc → a → b → b}
```

Then, we modify the product case (and others) with an associativity argument.

```
instance Generic (Crush b) where
  ...
  rprod ra rb = Crush go
    where
      go s@AssocLeft  (a :*: b) = selCrush rb s b · selCrush ra s a
      go s@AssocRight (a :*: b) = selCrush ra s a · selCrush rb s b
  ...
```

Defining Crush (10)

We have defined the "core" generic function, so now we can define our user-friendly wrapper.

$$\mathsf{crush} :: (\ldots) \Rightarrow \mathsf{Assoc} \to (\mathsf{a} \to \mathsf{b} \to \mathsf{b}) \to \mathsf{b} \to \mathsf{f}\ \mathsf{a} \to \mathsf{b}$$

Let's first review the definitions we've collected.

$$\begin{aligned}
&\mathsf{Crush} &&:: (\mathsf{Assoc} \to \mathsf{a} \to \mathsf{b} \to \mathsf{b}) \to \mathsf{Crush}\ \mathsf{b}\ \mathsf{a} \\
&\mathsf{frep} &&:: (\mathsf{FRep}\ \mathsf{g}\ \mathsf{f}) \Rightarrow \mathsf{g}\ \mathsf{a} \to \mathsf{g}\ (\mathsf{f}\ \mathsf{a}) \\
&\mathsf{selCrush} &&:: \mathsf{Crush}\ \mathsf{b}\ \mathsf{a} \to \mathsf{Assoc} \to \mathsf{a} \to \mathsf{b} \to \mathsf{b}
\end{aligned}$$

Notice any patterns?

Defining Crush (11)

First, we need a higher-order combining function.

$$\mathsf{Crush} :: (\mathsf{Assoc} \to a \to b \to b) \to \mathsf{Crush}\ b\ a$$

Next, we need to transform the generic type parameter a to a functional kind using the new representation dispatcher.

$$\mathsf{frep} :: (\mathsf{FRep}\ g\ f) \Rightarrow g\ a \to g\ (f\ a)$$

$$\begin{aligned}\mathsf{frep} \cdot \mathsf{Crush} :: {} & (\mathsf{FRep}\ (\mathsf{Crush}\ b)\ f) \Rightarrow \\ & (\mathsf{Assoc} \to a \to b \to b) \to \mathsf{Crush}\ b\ (f\ a)\end{aligned}$$

Then, we open the Crush value to get the generic function.

$$\mathsf{selCrush} :: \mathsf{Crush}\ b\ a \to \mathsf{Assoc} \to a \to b \to b$$

$$\begin{aligned}& \mathsf{selCrush} \cdot \mathsf{frep} \cdot \mathsf{Crush} \\ & \quad :: (\mathsf{FRep}\ (\mathsf{Crush}\ b)\ f) \Rightarrow \\ & \qquad (\mathsf{Assoc} \to a \to b \to b) \to \mathsf{Assoc} \to f\ a \to b \to b\end{aligned}$$

Defining Crush (12)

Finally, with a little massaging, we can define crush.

$$\begin{aligned}&\mathsf{crush} :: (\mathsf{FRep}\ (\mathsf{Crush\ b})\ \mathsf{f}) \Rightarrow\\ &\qquad \mathsf{Assoc} \to (\mathsf{a} \to \mathsf{b} \to \mathsf{b}) \to \mathsf{b} \to \mathsf{f\ a} \to \mathsf{b}\\ &\mathsf{crush\ s\ f\ z\ x} = \mathsf{selCrush}\ (\mathsf{frep}\ (\mathsf{Crush}\ (\mathsf{const\ f})))\ \mathsf{s\ x\ z}\end{aligned}$$

And we can define more wrappers that imply the associativity.

$$\begin{aligned}&\mathsf{crushl}, \mathsf{crushr} :: (\mathsf{FRep}\ (\mathsf{Crush\ b})\ \mathsf{f}) \Rightarrow\\ &\qquad (\mathsf{a} \to \mathsf{b} \to \mathsf{b}) \to \mathsf{b} \to \mathsf{f\ a} \to \mathsf{b}\\ &\mathsf{crushl} = \mathsf{crush\ AssocLeft}\\ &\mathsf{crushr} = \mathsf{crush\ AssocRight}\end{aligned}$$

That's it for the generic function definition, but what's the point? What can we do with Crush?

Using Crush (1)

Due to its genericity, Crush is a powerful and practical function. We can build a large number of functions using crush.

- Flatten a container to a list of its elements:

  $\mathsf{flattenr} :: (\mathsf{FRep}\ (\mathsf{Crush}\ [\mathsf{a}])\ \mathsf{f}) \Rightarrow \mathsf{f}\ \mathsf{a} \rightarrow [\mathsf{a}]$
  $\mathsf{flattenr} = \mathsf{crushr}\ (:)\ [\,]$
  $\mathsf{test2} = \mathsf{flattenr}\ (\mathsf{Node}\ 2\ (\mathsf{Leaf}\ \texttt{"Hi"})\ (\mathsf{Leaf}\ \texttt{"London"}))$
  $\quad == [\texttt{"Hi"}, \texttt{"London"}]$

- Or extract the reversed list:

  $\mathsf{flattenl} :: (\mathsf{FRep}\ (\mathsf{Crush}\ [\mathsf{a}])\ \mathsf{f}) \Rightarrow \mathsf{f}\ \mathsf{a} \rightarrow [\mathsf{a}]$
  $\mathsf{flattenl} = \mathsf{crushl}\ (:)\ [\,]$
  $\mathsf{test3} = \mathsf{flattenl}\ (\mathsf{Node}\ 2009\ (\mathsf{Leaf}\ 7)\ (\mathsf{Leaf}\ 9)) == [9, 7]$

Notice the use of associativity.

Using Crush (2)

- Sum the (numerical) elements of a container:

$$\mathsf{sum} :: (\mathsf{Num\ a}, \mathsf{FRep\ (Crush\ a)\ f}) \Rightarrow \mathsf{f\ a} \rightarrow \mathsf{a}$$
$$\mathsf{sum} = \mathsf{crushr}\ (+)\ 0$$
$$\mathsf{test4} = \mathsf{sum\ (Node\ 4\ (Leaf\ 40)\ (Leaf\ 2))} == 42$$

- Or determine if any element satisfies a predicate:

$$\mathsf{any} :: (\mathsf{FRep\ (Crush\ Bool)\ f}) \Rightarrow (\mathsf{a} \rightarrow \mathsf{Bool}) \rightarrow \mathsf{f\ a} \rightarrow \mathsf{Bool}$$
$$\mathsf{any\ p} = \mathsf{crushr}\ (\lambda \mathsf{x\ b} \rightarrow \mathsf{b} \vee \mathsf{p\ x})\ \mathsf{False}$$
$$\mathsf{test5} = \mathsf{any}\ (>2)\ \mathsf{(Node\ 5\ (Leaf\ 0)\ (Leaf\ 1))} == \mathsf{False}$$

The Crush function and its many derivatives are all available in the `emgm` package.

Diversion: Ad Hoc Instances (1)

Let's deviate from defining generic functions for a bit and explore why EMGM is extensible and modular. The reason is that we can override how a generic function works for any datatype. The mechanism is called an **ad hoc instance**.

Suppose we want to change the "empty" value for Tree Char. The generic result, as we have seen, is Tip, but we want something different.

```
instance Rep Empty (Tree Char) where
  rep = Empty (Leaf empty)

test6 = empty == Leaf '\NUL'
```

The instance specifies the function signature, Empty, and the type for the instance, Tree Char.

Note that you must have overlapping instances enabled for ad hoc instances:

```
{-# LANGUAGE OverlappingInstances #-}
```

Ad Hoc Instances (2)

The example of Empty is simple to understand, but it does not do justice to the flexibility that ad hoc instances provide.

Functions such as the Read and Show are very suitable for ad hoc instances. Indeed, the `emgm` package uses them to support the special syntax for lists and tuples.

```
instance (Rep Read a) ⇒ Rep Read [a] where
  rep = Read $ const $ list $ readPrec

instance (Rep Show a, Rep Show b) ⇒ Rep Show (a, b) where
  rep = Show s
    where s _ _ (a, b) = showTuple [shows a, shows b]
```

Return from Diversion: Defining Collect (1)

The last function we will define is also a useful one and takes full advantage of ad hoc instances.

The purpose of Collect is to gather all (top-level) values of a certain type from a value of a (different) type and return them in a list. Collect relies on a simple ad hoc instance for each type to match the collected value with the result value.

The function signature is:

$$\textbf{newtype}\ \mathsf{Collect}\ \mathsf{b}\ \mathsf{a} = \mathsf{Collect}\ \{\mathsf{selCollect} :: \mathsf{a} \to [\mathsf{b}]\}$$

The type parameter a represents the generic collected type, and b represents the non-generic result type.

Defining Collect (2)

Now, onto the definition.

As with Crush, the constant types (including Unit) and the cases for constructors and types are quite straightforward.

```
instance Generic (Collect b) where
  rconstant        = Collect (const [])
  rcon  cd      ra = Collect (selCollect ra)
  rtype td ep ra   = Collect (selCollect ra · from ep)
```

The key to keep in mind with this generic function is that the structural induction simply recurses throughout the value. It is the ad hoc instances that do the important work.

Defining Collect (3)

The sum case recursively dives into the indicated alternative.

```
rsum ra rb = Collect go
  where go (L a) = selCollect ra a
        go (R b) = selCollect rb b
```

The product case appends the collected results of one component to those of the other.

```
rprod ra rb = Collect go
  where go (a :*: b) = selCollect ra a ++ selCollect rb b
```

The wrapper itself is quite simple.

```
collect :: (Rep (Collect b) a) => a -> [b]
collect = selCollect rep
```

Defining Collect (4)

So, what about the ad hoc instances? Here is an example:

```
instance Rep (Collect Int) Int where
  rep = Collect (:[])
```

And here's another:

```
instance Rep (Collect (Tree a)) (Tree a) where
  rep = Collect (:[])
```

(And guess what? This is generated by `$(derive '' Tree)`!)

Using Collect

The function collect is easy to use...

```
val1 = Node 88 (Leaf 'a') (Leaf 'b')
test7 = collect val1 == "ab"
test8 = collect val1 == [88 :: Int]
```

... as long as you remember that the result type must be non-polymorphic and unambiguous.

```
val2 :: Tree Int
val2 = (Node 1 (Node 2 (Leaf 3) (Leaf 4)) (Leaf 5))
test9 = collect val2 == [1, 2, 3, 4, 5 :: Int]
test10 = collect val2 == [val2]
```

Looking at `emgm`

We have discussed how to write several generic functions. If you set off to implement your own, you should have a good idea of where to start.

If you instead want to simply use the available generic functions in the `emgm` package, you should look at the Haddock docs:

`http://hackage.haskell.org/package/emgm/`

(Yes, look at it now...)

Continuing Development of EMGM

EMGM is continuing to evolve. We have plans for a number of new functions or packages.

- $\mathsf{transpose} :: \mathsf{f}\ (\mathsf{g}\ \mathsf{a}) \to \mathsf{g}\ (\mathsf{f}\ \mathsf{a})$
- Map with first-class higher-order generic function
- Supporting `binary`, `bytestring`, `HDBC`

Cheers!

We would also be happy to take bug reports, contributions, or see other packages using `emgm`.

All roads to more information start at the homepage.

`http://www.cs.uu.nl/wiki/GenericProgramming/EMGM`

(I hope you don't hit the London traffic on your way there.)