# A Generic Deriving Mechanism for Haskell

José Pedro Magalhães[1] Atze Dijkstra[1] Johan Jeuring[1,2] Andres Löh[1]

[1]Department of Information and Computing Sciences, Utrecht University, P.O. Box 80.089, 3508 TB Utrecht, The Netherlands
[2]School of Computer Science, Open University of the Netherlands, P.O. Box 2960, 6401 DL Heerlen, The Netherlands

{jpm,atze,johanj,andres}@cs.uu.nl

## Abstract

Haskell's **deriving** mechanism supports the automatic generation of instances for a number of functions. The Haskell 98 Report only specifies how to generate instances for the *Eq*, *Ord*, *Enum*, *Bounded*, *Show*, and *Read* classes. The description of how to generate instances is largely informal. The generation of instances imposes restrictions on the shape of datatypes, depending on the particular class to derive. As a consequence, the portability of instances across different compilers is not guaranteed.

We propose a new approach to Haskell's **deriving** mechanism, which allows users to specify how to derive arbitrary class instances using standard datatype-generic programming techniques. Generic functions, including the methods from six standard Haskell 98 derivable classes, can be specified entirely within Haskell 98 plus multi-parameter type classes, making them lightweight and portable. We can also express *Functor*, *Typeable*, and many other derivable classes with our technique. We implemented our **deriving** mechanism together with many new derivable classes in the Utrecht Haskell Compiler.

***Categories and Subject Descriptors*** D.1.1 [*Programming Techniques*]: Functional Programming

***General Terms*** Languages

## 1. Introduction

Generic programming has come a long way: from its roots in category theory (Backhouse et al. 1999), passing through dedicated languages (Jansson and Jeuring 1997), language extensions and pre-processors (Hinze et al. 2007; Löh 2004) until the flurry of library-based approaches of today (Rodriguez Yakushev et al. 2008). In this evolution, expressivity has not always increased: many generic programming libraries of today still cannot compete with the Generic Haskell pre-processor, for instance. The same applies to performance, as libraries tend to do little regarding code optimization, whereas meta-programming techniques such as Template Haskell (Sheard and Peyton Jones 2002) can generate near-optimal code. Instead, generic programming techniques seem to evolve in the direction of better availability and usability: it should be easy to define generic functions and it should be trivial to use them. Certainly some of the success of the Scrap Your Boilerplate approach (SYB, Lämmel and Peyton Jones 2003, 2004) is due to its availability: it comes with the Glasgow Haskell Compiler (GHC), the main Haskell compiler, which can even derive the necessary type class instances to make everything work without clutter.

To improve the usability of generics in Haskell, we believe a tighter integration with the compiler is necessary. In fact, the Haskell 98 standard already contains some generic programming, in the form of derived instances (Peyton Jones et al. 2003, Chapter 10). Unfortunately, the report does not formally specify how to derive instances, and it restricts the classes that can be derived to six only (*Eq*, *Ord*, *Enum*, *Bounded*, *Show*, and *Read*). GHC has since long extended these with *Data* and *Typeable* (the basis of SYB), and more recently with *Functor*, *Foldable* and *Traversable*. Due to the lack of a unifying formalism, these extensions are not easily mimicked in other compilers, which need to reimplement the instance code generation mechanism.

To address these issues, we propose an approach to specifying how to derive an instance of a class, together with new behavior for the **deriving** mechanism in Haskell to automatically derive such a class. To allow for portability across compilers, our approach requires only Haskell 98 with multi-parameter type classes and support for a new compiler pragma. Specifically, our contributions are:

- We describe a new datatype-generic programming library for Haskell. Although similar in many aspects to other approaches, our library requires almost no extensions to Haskell 98; the most significant requirement is support for multi-parameter type classes.
- We show how this library can be used to extend the **deriving** mechanism in Haskell, and provide sample derivings, notably for the *Functor* class.
- We provide a detailed description of how the representation for a datatype is generated. In particular, we can represent almost all Haskell 98 datatypes.
- We provide a fully functional implementation of our library in the Utrecht Haskell Compiler (UHC, Dijkstra et al. 2009). Many useful generic functions are defined using generic deriving in the compiler.

We also provide a package which compiles both in UHC and GHC, showing in detail the code that needs to added to the compiler, the code that should be generated by the compiler, and the code that is portable between compilers.[1]

The remainder of this paper is structured as follows: first we give a brief introduction to generic programming in Haskell (Section 2), which also introduces the particular library we use. We proceed to show how to define generic functions (Section 3), and then

[1] `http://dreixel.net/research/code/gdmh.tar.gz`



*Haskell'10,* September 30, 2010, Baltimore, Maryland, USA.

describe the necessary modifications to the compiler for supporting our approach (Section 4). Finally, we discuss alternative designs (Section 5), review related work (Section 6), propose future work (Section 7) and conclude in Section 8.

## 2. Generic programming

We use the generic function *encode* as a running example throughout this paper. This function transforms a value into a sequence of bits:

```
data Bit = 0 | 1
class Encode α where
  encode :: α → [Bit]
```

We want the user to be able to write

```
data Exp = Const Int | Plus Exp Exp
  deriving (Show, Encode)
```

and to use *encode* like

```
test :: [Bit]
test = encode (Plus (Const 1) (Const 2))
```

This should be all that is necessary to use *encode*. The user should need no further knowledge of generics, and *encode* can be used in the same way as *show*, for instance.

Behind the scenes, the compiler generates an instance for *Encode Exp* based on a generic specification of instances of class *Encode*. There are several ways to specify such an instance, both using code generation and datatype-generic approaches. We choose a datatype-generic approach because it is type-safe and elegant (Hinze et al. 2007). We will discuss alternative designs and motivate our choice in more detail in Section 5. For now we proceed to describe our new generic programming library. The three basic ingredients for generic programming, as described by Hinze and Löh (2009), are:

1. Support for overloaded functions
2. A run-time type representation
3. A generic view on data

Since we use Haskell, (1) is easy: an overloaded (ad-hoc polymorphic) function is a method of a type class. For (2), we introduce a type representation similar to the one used in the `regular` (Van Noort et al. 2008) and `instant-generics` (Chakravarty et al. 2009) libraries, in Section 2.1. For (3), we again use type classes to encode embedding-projection pairs for user-defined datatypes in Section 2.3.

### 2.1 A run-time type representation

The choice of a run-time type representation affects not only the compiler writer but also the expressiveness of the whole approach. A simple representation is easier to derive, but might not allow the definition of some generic functions. More complex representations are more expressive, but require more work for the automatic derivation of instances.

We present a set of representation types that tries to balance these factors. We use the common sum-of-products representation without explicit fixpoints but with explicit abstraction over a single parameter. Therefore, representable types are functors, and we can compose types. Additionally, we provide useful types for encoding meta-information (such as constructor names) and tagging arguments to constructors. We show examples of how these representation types are used in Section 2.4.

The basic ingredients of the sum-of-products representation types are:

```
data U1      ρ = U1
data (+) φ ψ ρ = L1 {unL1 :: φ ρ} | R1 {unR1 :: ψ ρ}
data (×) φ ψ ρ = φ ρ × ψ ρ
```

We encode lifted sums with $(+)$ and lifted products with $(\times)$. Nullary products are encoded with lifted unit ($U_1$).[2]

The type variable $\rho$ is present in all representation types: it represents the parameter over which we abstract. We use an explicit combinator to mark the occurrence of this parameter:

```
newtype Par1 ρ = Par1 {unPar1 :: ρ}
```

As our representation is functorial, we can encode composition. Although we cannot express this in the kind system, we require the first argument of composition to be a representable type constructor. The second argument can only be the parameter, a recursive occurrence of a functorial datatype, or again a composition. We use $Rec_1$ to represent recursion, and $(\circ)$ for composition:

```
newtype Rec1 φ    ρ = Rec1 {unRec1 :: φ ρ}
newtype (∘)  φ ψ ρ = Comp1 (φ (ψ ρ))
```

PolyP (Jansson and Jeuring 1997) treats composition in a similar way.

Finally, we have two types for representing meta-information and tagging:

```
newtype K1 ι γ    ρ = K1 {unK1 :: γ}
newtype M1 ι γ φ ρ = M1 {unM1 :: φ ρ}
```

We use $K_1$ for tagging and $M_1$ for storing meta-information. The role of the $\iota$ parameter in these types is made explicit by the following type synonyms:

```
data D                type D1    = M1 D
data C                type C1    = M1 C
data S                type S1    = M1 S
data R                type Rec0  = K1 R
data P                type Par0  = K1 P
```

We use $Rec_0$ to tag occurrences of (possibly recursive) types of kind $\star$ and $Par_0$ to mark additional parameters of kind $\star$ (other than $\rho$). For meta-information, we use $D_1$ for datatype information, $C_1$ for constructor information and $S_1$ for record selector information. We group five combinators into two because in many generic functions the behavior is independent of the meta-information or tags. In this way, fewer trivial cases have to be given. We present the meta-information associated with $M_1$ in detail in the next section.

Note that we abstract over a single parameter $\rho$ of kind $\star$. This means we will be able to express generic functions such as

$$fmap :: (\alpha \to \beta) \to \phi\ \alpha \to \phi\ \beta$$

but not

$$bimap :: (\alpha \to \gamma) \to (\beta \to \delta) \to \phi\ \alpha\ \beta \to \phi\ \gamma\ \delta$$

For *bimap* we need another type representation that can distinguish between the parameters. All representation types need to carry one additional type argument. However, we think that, in practice, few generic functions require abstraction over more than a single type parameter.

### 2.2 Meta-information

For some generic functions we need information about datatypes, constructors, and records. This information is stored in the type representation:

[2] We also have lifted void ($V_1$) to represent nullary sums, but for simplicity we omit it from this discussion and from the generic functions in Section 3.

```
class Datatype γ where
  datatypeName :: γ → String
  moduleName   :: γ → String

class Selector γ where
  selName :: γ → String
  selName = const ""

class Constructor γ where
  conName :: γ → String
  conFixity :: γ → Fixity
  conFixity = const Prefix
  conIsRecord :: γ → Bool
  conIsRecord = const False
```

Names are unqualified. We provide the datatype name together with the module name. This is the only meta-information we store for a datatype, although it could be easily extended to add the kind, for example. We only store the name of a selector. For a constructor, we also store its fixity and mark if it has fields. This last information is not strictly necessary, as it can be inferred by looking for non-empty *selName*s, but it simplifies some generic function definitions. The datatypes *Fixity* and *Associativity* are unsurprising:

```
data Fixity = Prefix | Infix Associativity Int
data Associativity = LeftAssociative | RightAssociative
                   | NotAssociative
```

We provide default definitions for *conFixity* and *conIsRecord* to simplify instantiation for prefix constructors that do not use record notation.[3]

Finally, we tie the meta-information to the representation:

```
instance (Datatype γ) ⇒ Datatype (M1 D γ ϕ ρ) where
  datatypeName = datatypeName ∘ unMeta
  moduleName   = moduleName   ∘ unMeta
instance (Constructor γ) ⇒ Constructor (M1 C γ ϕ ρ) where
  conName = conName ∘ unMeta
instance (Selector γ) ⇒ Selector (M1 S γ ϕ ρ) where
  selName  = selName  ∘ unMeta
unMeta :: M1 ι γ ϕ ρ → γ
unMeta = ⊥
```

Function *unMeta* operates at the type-level only, so it does not need an implementation. We provide more details in Section 4.5, and the examples later in Section 2.4 and Section 3.6 also clarify how we use these classes.

Note that we could encode the meta information as an extra argument to $M_1$:

```
data M1 ι γ ϕ ρ = M1 Meta (ϕ ρ)
data Meta = Meta String Fixity ...
```

However, with this encoding we have trouble writing generic producers, since when we are producing an $M_1$ we have to produce a *Meta* for which we have no information. With the above representation we avoid this problem by using type-classes to fill in the right information for us. See Section 3.5 for an example of how this works.

[3] We also provide an empty default *selName* because all constructor arguments will be wrapped in an $S_1$, independently of using record notation or not. We omit this in the example representations of this section for space reasons, but it becomes clear in Section 4.

### 2.3 A generic view on data

We obtain a generic view on data by defining an embedding-projection pair between a datatype and its type representation. We use the following classes for this purpose:

```
class Representable0 α τ where
  from0 :: α → τ χ
  to0   :: τ χ → α
class Representable1 ϕ τ where
  from1 :: ϕ ρ → τ ρ
  to1   :: τ ρ → ϕ ρ
```

We use $\tau$ to encode the representation of a standard type. Since $\tau$ is built from representation types, it is functorial. In *Representable*$_1$, we encode types of kind $\star \to \star$, so we have the parameter $\rho$. In *Representable*$_0$ there is no parameter, so we invent a variable $\chi$ which is never used.

All types need to have an instance of *Representable*$_0$. Types of kind $\star \to \star$ also need an instance of *Representable*$_1$. This separation is necessary because some generic functions (like *fmap* or *traverse*) require explicit abstraction from a single type parameter, whereas others (like *show* or *enum*) do not. Given the different kinds involved, it is unavoidable to have two type classes for this representation. Note, however, that we have a single set of representation types (apart from the duplication for tagging recursion and parameters).

***Avoiding extensions*** Since we want to avoid using advanced Haskell extensions such as type families (Schrijvers et al. 2008) or functional dependencies (Jones 2000), we use a simple multi-parameter type class for embedding-projection pairs. In fact, $\tau$ is uniquely determined by $\alpha$ (and $\phi$). We could encode the representation type more naturally with a type family:

```
class Representable0 α where
  type Rep0 α :: ⋆ → ⋆
  from0 :: α → Rep0 α χ
  to0   :: Rep0 α χ → α
```

Since type families and functional dependencies are not yet part of any Haskell standard, we do not use them. Instead, we use multi-parameter type classes, and solve the ambiguities that arise by coercing with *asTypeOf*.

### 2.4 Example representations

We now show how to represent some standard datatypes. Note that all the code in this section is automatically generated by the compiler, as described in Section 4.

***Representing Exp.*** The meta-information for datatype *Exp* looks as follows:

```
data $Exp
data $Const_Exp
data $Plus_Exp
instance Datatype $Exp where
  moduleName   _ = "ModuleName"
  datatypeName _ = "Exp"
instance Constructor $Const_Exp where conName _ = "Const"
instance Constructor $Plus_Exp  where conName _ = "Plus"
```

In *moduleName*, `"ModuleName"` is the name of the module where *Exp* lives. The particular datatypes we use for representing the meta-information at the type-level are not needed for defining generic functions, so they are not visible to the user. In this paper, we prefix them with a \$.

The type representation ties the meta-information to the sum-of-products representation of *Exp*:

```
type Rep_0^Exp =
  D_1 $Exp  ( C_1 $Const_Exp (Rec_0 Int)
            + C_1 $Plus_Exp  (Rec_0 Exp × Rec_0 Exp))
```

Note that the representation is shallow: at the recursive occurrences we use *Exp*, and not $Rep_0^{Exp}$.

The embedding-projection pair implements the isomorphism between *Exp* and $Rep_0^{Exp}$:

```
instance Representable_0 Exp Rep_0^Exp where
  from_0 (Const n)   = M_1 (L_1 (M_1 (K_1 n)))
  from_0 (Plus e e') = M_1 (R_1 (M_1 (K_1 e × K_1 e')))
  to_0 (M_1 (L_1 (M_1 (K_1 n))))          = Const n
  to_0 (M_1 (R_1 (M_1 (K_1 e × K_1 e')))) = Plus e e'
```

Here it is clear that $from_0$ and $to_0$ are inverses: the pattern of $from_0$ is the same as the expression in $to_0$, and vice-versa.

***Representing lists.*** The representation for a type of kind $\star \to \star$ requires an instance for both $Representable_1$ and $Representable_0$. For lists

```
data List ρ = Nil | Cons ρ (List ρ) deriving (Show, Encode)
```

we generate the following code:

```
type Rep_0^List ρ =
  D_1 $List  ( C_1 $Nil_List   U_1
             + C_1 $Cons_List  (Par_0 ρ × Rec_0 (List ρ)))

instance Representable_0 (List ρ) (Rep_0^List ρ) where
  from_0 Nil       = M_1 (L_1 (M_1 U_1))
  from_0 (Cons h t) = M_1 (R_1 (M_1 (K_1 h × K_1 t)))
  to_0 (M_1 (L_1 (M_1 U_1)))          = Nil
  to_0 (M_1 (R_1 (M_1 (K_1 h × K_1 t)))) = Cons h t
```

We omit the definitions for the meta-information, which are similar to the previous example. We use $Par_0$ to tag the parameter $\rho$, as we view lists as a kind $\star$ datatype for $Representable_0$. This is different in the $Representable_1$ instance:

```
type Rep_1^List = D_1 $List  ( C_1 $Nil_List   U_1
                             + C_1 $Cons_List  (Par_1 × Rec_1 List))

instance Representable_1 List Rep_1^List where
  from_1 Nil       = M_1 (L_1 (M_1 U_1))
  from_1 (Cons h t) = M_1 (R_1 (M_1 (Par_1 h × Rec_1 t)))
  to_1 (M_1 (L_1 (M_1 U_1)))          = Nil
  to_1 (M_1 (R_1 (M_1 (Par_1 h × Rec_1 t)))) = Cons h t
```

We treat parameters and recursion differently in $Rep_0^{List}$ and $Rep_1^{List}$. In $Rep_0^{List}$ we use $Par_0$ and $Rec_0$ for mere tagging; in $Rep_1^{List}$ we use $Par_1$ and $Rec_1$ instead, which store the parameter and the recursive occurrence of a type constructor, respectively. We will see later when defining generic functions (Section 3) how these are used.

***Representing type composition.*** We now present a larger example, involving more complex datatypes, to show the expressiveness of our approach. Datatype *Expr* represents abstract syntax trees of a small language:

```
infixr 6 ∗
data Expr ρ = Const Int
            | Expr ρ ∗ Expr ρ
            | Var_Expr {unVar :: Var ρ}
            | Let [Decl ρ] (Expr ρ)
data Decl ρ = Decl (Var ρ) (Expr ρ)
data Var  ρ = Var ρ | Var_L (Var [ρ])
```

Note that *Expr* makes use of an infix constructor ($*$), has a selector (*unVar*), and uses lists in *Let*. Datatype *Var* is nested, since in the $Var_L$ constructor *Var* is called with $[\rho]$. These oddities are present only for illustrating how our approach represents them. We show only the essentials of the encoding of this set of mutually recursive datatypes, starting with the meta-information:

```
data $Times_Expr
data $Var_Expr_Expr
data $UnVar

instance Constructor $Times_Expr where
  conName     _ = "*"
  conFixity   _ = Infix RightAssociative 6
instance Constructor $Var_Expr_Expr where
  conName     _ = "Var_Expr"
  conIsRecord _ = True
instance Selector $UnVar where selName _ = "unVar"
```

We have to store the fixity of the $*$ constructor, and also the fact that $Var_{Expr}$ has a record. We store its name in the instance for *Selector*, and tie the meta-information to the representation:

```
type Rep_1^Expr = D_1 $Expr
  (  (  C_1 $Const_Expr   (Rec_0 Int)
      + C_1 $Times_Expr   (Rec_1 Expr × Rec_1 Expr))
   + (  C_1 $Var_Expr_Expr (S_1 $UnVar (Rec_1 Var))
      + C_1 $Let_Expr     (([] ∘ Rec_1 Decl) × Rec_1 Expr)))
```

In $Rep_1^{Expr}$ we see the use of $S_1$. Also interesting is the representation of the *Let* constructor: the list datatype is applied not to the parameter $\rho$ but to *Decl* $\rho$, so we use composition to denote this. Note also that we are using a balanced encoding for the sums (and also for the products). This improves the performance of the type-checker, and makes generic encoding more space-efficient, for instance.

We omit the representation for *Decl*. For *Var* we use composition again:

```
type Rep_1^Var = D_1 $Var
                 (     C_1 $Var_Var   Par_1
                     + C_1 $Var_L_Var (Var ∘ Rec_1 []))
```

In the $Var_L$ constructor, *Var* is applied to $[\rho]$. We represent this as a composition with $Rec_1$ [].

When we use composition, the embedding-projection pairs become slightly more complicated:

```
instance Representable_1 Expr Rep_1^Expr where
  from_1 (Const i)   = M_1 (L_1 (L_1 (M_1 (K_1 i))))
  from_1 (e_1 ∗ e_2) = M_1 (L_1 (R_1 (M_1 (Rec_1 e_1 × Rec_1 e_2))))
  from_1 (Var_Expr v) = M_1 (R_1 (L_1 (M_1 (M_1 (Rec_1 v)))))
  from_1 (Let d e)   =
    M_1 (R_1 (R_1 (M_1 (Comp_1 (fmap Rec_1 d) × Rec_1 e))))
  to_1 (M_1 (L_1 (L_1 (M_1 (K_1 i)))))          = Const i
  to_1 (M_1 (L_1 (R_1 (M_1 (Rec_1 e_1 × Rec_1 e_2))))) = e_1 ∗ e_2
  to_1 (M_1 (R_1 (L_1 (M_1 (M_1 (Rec_1 v)))))) = Var_Expr v
  to_1 (M_1 (R_1 (R_1 (M_1 (Comp_1 d × Rec_1 e))))) =
    Let (fmap unRec_1 d) e
```

We need to use *fmap* to apply the $Rec_1$ constructor inside the lists. In this case we could use *map* instead, but in general we require the first argument to $\circ$ to have a *Functor* instance so we can use *fmap*. In $to_1$ we need to convert back, this time mapping $unRec_1$.

For *Var*, the embedding-projection pair is similar:

```
instance Representable1 Var Rep1^Var where
  from1 (Var x)    = M1 (L1 (M1 (Par1 x)))
  from1 (Var_L xs) = M1 (R1 (M1 (Comp1 (fmap Rec1 xs))))
  to1 (M1 (L1 (M1 (Par1 x))))      = Var x
  to1 (M1 (R1 (M1 (Comp1 xs))))) = Var_L (fmap unRec1 xs)
```

Note that composition is used both in the representation for the first argument of constructor *Let* (of type $[Decl\ \rho]$) and in the nested recursion of $Var_L$ (of type $Var\ [\rho]$). In both cases, we have a recursive occurrence of a parametrized datatype where the parameter is not just the variable $\rho$. Recall our definition of composition:

```
data (∘) φ ψ ρ = Comp1 (φ (ψ ρ))
```

The type $\phi$ is applied not to $\rho$, but to the result of applying $\psi$ to $\rho$. This is why we use $\circ$ when the recursive argument to a datatype is not $\rho$, like in $[Decl\ \rho]$ and $Var\ [\rho]$. When it is $\rho$, we can simply use $Rec_1$.

We have seen how to represent many features of Haskell datatypes in our approach. We give a detailed discussion of the supported datatypes in Section 7.1.

## 3. Generic functions

In this section we show how to define type classes with derivable functions.

### 3.1 Generic function definition

Function *encode* is a method of a type-class:

```
data Bit = 0 | 1

class Encode α where
  encode :: α → [Bit]
```

We cannot provide instances of *Encode* for our representation types, as those have kind $\star \to \star$, and *Encode* expects a parameter of kind $\star$. We therefore define a helper class, this time parametrized over a variable of kind $\star \to \star$:

```
class Encode1 φ where
  encode1 :: φ χ → [Bit]
```

For constructors without arguments we return the empty list, as there is nothing to encode. Meta-information is discarded:

```
instance Encode1 U1 where
  encode1 _ = []
instance (Encode1 φ) ⇒ Encode1 (M1 ι γ φ) where
  encode1 (M1 a) = encode1 a
```

For a value of a sum type we produce a single bit to record the choice. For products we concatenate the encoding of each element:

```
instance (Encode1 φ, Encode1 ψ) ⇒ Encode1 (φ + ψ) where
  encode1 (L1 a) = 0 : encode1 a
  encode1 (R1 a) = 1 : encode1 a
instance (Encode1 φ, Encode1 ψ) ⇒ Encode1 (φ × ψ) where
  encode1 (a × b) = encode1 a ++ encode1 b
```

It remains to encode constants. Since constant types have kind $\star$, we resort to *Encode*:

```
instance (Encode φ) ⇒ Encode1 (K1 ι φ) where
  encode1 (K1 a) = encode a
```

Note that while the instances for the representation types are given for the $Encode_1$ class, only the *Encode* class is exported and allowed to be derived. This is because its type is more general, and because we need a two-level approach to deal with recursion: for the $K_1$ instance, we recursively call *encode* instead of $encode_1$. Recall our representation for *Exp* (simplified and with type synonyms expanded):

```
type Rep0^Exp = K1 R Int + K1 R Exp × K1 R Exp
```

Since *Int* and *Exp* appear as arguments to $K_1$, and our instance of $Encode_1$ for $K_1\ \iota\ \phi$ requires an instance of $Encode\ \phi$, we need instances of *Encode* for *Int* and for *Exp*. We deal with *Int* in the next section, and *Exp* in Section 3.3. Finally, note that we do not need $Encode_1$ instances for $Rec_1$, $Par_1$ or $(\circ)$. These are only required for generic functions which make use of the $Representable_1$ class. We will see an example in Section 3.4.

### 3.2 Base types

We have to provide the instances of *Encode* for the base types:

```
instance Encode Int  where encode = ...
instance Encode Char where encode = ...
```

Since *Encode* is exported, a user can also provide additional base type instances, or ad-hoc instances (types for which the required implementation is different from the derived generic behavior).

### 3.3 Default definition

We miss an instance of *Encode* for *Exp*. Instances of generic functions for representable types rely on the embedding-projection pair to convert from/to the type representation and then apply the generic function:

```
encode_Default :: (Representable0 α τ, Encode1 τ)
                  ⇒ τ χ → α → [Bit]
encode_Default rep x = encode1 ((from0 x) `asTypeOf` rep)
```

Function $encode_{Default}$ tells the compiler what to fill in for the instance of each of the derived types. Because we do not want to use functional dependencies for portability reasons, we pass the representation type explicitly to function $encode_{Default}$. This function uses the representation type to coerce the result type of $from_0$ with *asTypeOf*. This slight complication is a small price to pay for extended portability.

Now we can show the instance of *Encode* for *Exp* and *List*:

```
instance Encode Exp where
  encode = encode_Default (⊥ :: Rep0^Exp χ)
instance (Encode ρ) ⇒ Encode (List ρ) where
  encode = encode_Default (⊥ :: Rep0^List ρ χ)
```

Both instances look similar and trivial. However, the instance for *List* requires scoped type variables to type-check. We can avoid the need for scoped type variables if we create an auxiliary local function $encode_{List}$ with the same type and behavior of $encode_{Default}$:

```
instance (Encode ρ) ⇒ Encode (List ρ) where
  encode = encode_List ⊥ where
    encode_List :: (Encode ρ) ⇒ Rep0^List ρ χ → List ρ → [Bit]
    encode_List = encode_Default
```

Here, the local function $encode_{List}$ encodes in its type the correspondence between the type $List\ \rho$ and its representation $Rep_0^{List}\ \rho$. Its type signature is required, but can easily be obtained from the type of $encode_{Default}$ by replacing the type variables $\alpha$ and $\tau$ with the concrete types for this instance.

For completeness, we give the instance for *Exp* in the same fashion:

```
instance Encode Exp where
  encode = encode_Exp ⊥ where
```

$$encode_{Exp} :: Rep_0^{Exp}\ \chi \to Exp \to [Bit]$$
$$encode_{Exp} = encode_{Default}$$

It might seem strange that we choose not to use Haskell's built-in functionality for default definitions for class methods. Unfortunately we cannot use default methods, for two reasons:

1. Since we avoid using type families and functional dependencies, we need to explicitly pass the representation type as an argument to $encode_{Default}$.
2. A default case would force us to move the $Representable_0\ \alpha\ \tau$ and $Encode_1\ \tau$ class constraints to the *Encode* class, possibly preventing ad-hoc instances for non-representable types and exposing $Encode_1$ to the user.

However, if the compiler is to generate instances for *Exp* and other representable datatypes automatically, how does it know which function to use as default? The alternative to standard Haskell default methods is to use a naming convention for this function (like appending *Default* to the class function name, as in our example). It is more reliable to use a pragma:

$\{-\#$ `DERIVABLE` *Encode encode* $encode_{Default}$ $\#-\}$

This pragma takes three arguments, which represent (respectively):

1. The class which we are defining as derivable
2. The method of the class which is generic (and therefore needs a default definition)
3. The name of the function which serves as a default definition

Such a pragma also has the advantage of indicating derivability for a particular class. We could use a keyword such as **derivable** to signal that a class is allowed to be derived:

**derivable class** *Encode* $\alpha$ **where** ...

However, by using a pragma instead (as described above) we ensure more portability, as compilers without support for our derivable type classes can still compile the code.

Since a class can have multiple generic methods, multiple pragmas can be used for this purpose. Note, however, that a derivable class can only have non-generic methods if there is a default definition for these, as otherwise we have no means for implementing the non-generic methods. Alternatively, we could treat generic methods as default methods, filling in the generic definition automatically if the user does not give a definition. This would allow classes to have normal, generic, and default methods. However, it would complicate the code generation mechanism.

### 3.4 Generic map

In this subsection we define the generic map function *fmap*, which implements the Prelude's *fmap*. Function *fmap* requires access to the parameter in the representation type. As before, we export a single class together with an internal class where we define the generic instances:

**class** *Functor* $\phi$ **where**
  $fmap :: (\rho \to \alpha) \to \phi\ \rho \to \phi\ \alpha$

**class** $Functor_1\ \phi$ **where**
  $fmap_1 :: (\rho \to \alpha) \to \phi\ \rho \to \phi\ \alpha$

Unlike in *Encode*, the type arguments to *Functor* and $Functor_1$ have the same kind, so we do not really need two classes. However, for consistency, we use the same style as for kind $\star$ generic functions.

We apply the argument function in the parameter case:

**instance** $Functor_1\ Par_1$ **where**
  $fmap_1\ f\ (Par_1\ a) = Par_1\ (f\ a)$

Unit and constant values do not change, as there is nothing we can map over. We apply $fmap_1$ recursively to meta-information, sums and products:

**instance** $Functor_1\ U_1$ **where**
  $fmap_1\ f\ U_1 \quad = U_1$

**instance** $Functor_1\ (K_1\ \iota\ \gamma)$ **where**
  $fmap_1\ f\ (K_1\ a) = K_1\ a$

**instance** $(Functor_1\ \phi) \Rightarrow Functor_1\ (M_1\ \iota\ \gamma\ \phi)$ **where**
  $fmap_1\ f\ (M_1\ a) = M_1\ (fmap_1\ f\ a)$

**instance** $(Functor_1\ \phi, Functor_1\ \psi) \Rightarrow Functor_1\ (\phi + \psi)$ **where**
  $fmap_1\ f\ (L_1\ a) = L_1\ (fmap_1\ f\ a)$
  $fmap_1\ f\ (R_1\ a) = R_1\ (fmap_1\ f\ a)$

**instance** $(Functor_1\ \phi, Functor_1\ \psi) \Rightarrow Functor_1\ (\phi \times \psi)$ **where**
  $fmap_1\ f\ (a \times b) = fmap_1\ f\ a \times fmap_1\ f\ b$

If we find a recursive occurrence of a functorial type, we call *fmap* again, to tie the recursive knot:

**instance** $(Functor\ \phi) \Rightarrow Functor_1\ (Rec_1\ \phi)$ **where**
  $fmap_1\ f\ (Rec_1\ a) = Rec_1\ (fmap\ f\ a)$

The remaining case is composition:

**instance** $(Functor\ \phi, Functor_1\ \psi) \Rightarrow Functor_1\ (\phi \circ \psi)$ **where**
  $fmap_1\ f\ (Comp_1\ x) = Comp_1\ (fmap\ (fmap_1\ f)\ x)$

Recall that we require the first argument of $(\circ)$ to be a user-defined datatype, and the second to be a representation type. Therefore, we use $fmap_1$ for the inner mapping (as it will map over a representation type) but *fmap* for the outer mapping (as it will require an embedding-projection pair). This is the general structure of the instance of $(\circ)$ for a generic function.

Finally, we define the default method:

$\{-\#$ `DERIVABLE` *Functor fmap* $fmap_{Default}$ $\#-\}$
$fmap_{Default} :: (Representable_1\ \phi\ \tau, Functor_1\ \tau)$
$\quad \Rightarrow \tau\ \rho \to (\rho \to \alpha) \to \phi\ \rho \to \phi\ \alpha$
$fmap_{Default}\ rep\ f\ x = to_1\ (fmap_1\ f\ (from_1\ x\ `asTypeOf`\ rep))$

Now *Functor* can be derived for user-defined datatypes. The usual restrictions apply: only types with at least one type parameter and whose last type argument is of kind $\star$ can derive *Functor*. The compiler derives the following instance for *List*:

**instance** *Functor List* **where**
  $fmap = fmap_{List}\ (\bot :: Rep_1^{List}\ \rho)$ **where**
    $fmap_{List} :: Rep_1^{List}\ \rho \to (\rho \to \alpha) \to List\ \rho \to List\ \alpha$
    $fmap_{List} = fmap_{Default}$

Note that the instance *Functor List* also guarantees that we can use *List* as the first argument to $(\circ)$, as the embedding-projection pairs for such compositions need to use *fmap*.

The instances derived for *Expr*, *Decl*, and *Var* are similar.

### 3.5 Generic empty

We can also easily express generic producers: functions which produce data. We will illustrate this with function *empty*, which produces a single value of a given type:

**class** *Empty* $\alpha$ **where** $empty :: \alpha$

This function is perhaps the simplest generic producer, as it consumes no data. It relies only on the structure of the datatype to produce values. Other examples of generic producers are the methods in *Read* and the *Arbitrary* class from `QuickCheck`, and `binary`'s *get*. As usual, we define an auxiliary type class:

```
class Empty1 φ where
  empty' :: φ χ
```

Most instances of $Empty_1$ are straightforward:

```
instance Empty1 U1 where
  empty' = U1
instance (Empty1 φ) ⇒ Empty1 (M1 ι γ φ) where
  empty' = M1 empty'
instance (Empty1 φ, Empty1 ψ) ⇒ Empty1 (φ × ψ) where
  empty' = empty' × empty'
instance (Empty φ) ⇒ Empty1 (K1 ι φ) where
  empty' = K1 empty
```

For units we can only produce $U_1$. Meta-information is produced with $M_1$, and since we encode the meta-information using type classes (instead of using extra arguments to $M_1$) we do not have to use $\perp$ here. An empty product is the product of empty components, and for $K_1$ we recursively call *empty*. The only interesting choice is for the sum type:

```
instance (Empty1 φ) ⇒ Empty1 (φ + ψ) where
  empty' = L1 empty'
```

In a sum, we always take the leftmost constructor for the empty value. Since the leftmost constructor might be recursive, function *empty* might not terminate. More complex implementations can look ahead to spot recursion, or choose alternative constructors after recursive calls, for instance. Note also the similarity between our *Empty* class and Haskell's *Bounded*: if we were defining *minBound* and *maxBound* generically, we could choose $L_1$ for *minBound* and $R_1$ for *maxBound*. This way we would preserve the semantics for derived *Bounded* instances, as defined by Peyton Jones et al. (2003), while at the same time lifting the restrictions on types that can derive *Bounded*. Alternatively, to keep the Haskell 98 behavior, we could give no instance for $\times$, as enumeration types will not have a product in their representations.

The default method simply applies $to_0$ to $empty'$:

```
{-# DERIVABLE Empty empty emptyDefault #-}
emptyDefault :: (Representable0 α τ, Empty1 τ)
             ⇒ τ χ → α
emptyDefault rep = to0 (empty' `asTypeOf` rep)
```

Now the compiler can produce instances such as:

```
instance Empty Exp where
  empty = emptyExp ⊥ where
    emptyExp :: Rep0Exp χ → Exp
    emptyExp = emptyDefault
instance (Empty ρ) ⇒ Empty (List ρ) where
  empty = emptyList ⊥ where
    emptyList :: (Empty ρ) ⇒ Rep0List ρ χ → List ρ
    emptyList = emptyDefault
```

Instances for other types are similar.

### 3.6 Generic show

To illustrate the use of constructor and selector labels, we define the *shows* function generically:

```
class Show α where
  shows :: α → ShowS
  show  :: α → String
  show x = shows x ""
```

We define a helper class $Show_1$, with $shows_1$ as the only method. For each representation type there is an instance of $Show_1$. The extra *Bool* argument will be explained later. Datatype meta-information and sums are ignored. For units we have nothing to show, and for constants we call *shows* recursively:

```
class Show1 φ where
  shows1 :: Bool → φ χ → ShowS

instance (Show1 φ) ⇒ Show1 (D1 γ φ) where
  shows1 b (M1 a) = shows1 b a
instance (Show1 φ, Show1 ψ) ⇒ Show1 (φ + ψ) where
  shows1 b (L1 a) = shows1 b a
  shows1 b (R1 a) = shows1 b a
instance Show1 U1 where
  shows1 _ U1     = id
instance (Show φ) ⇒ Show1 (K1 ι φ) where
  shows1 _ (K1 a) = shows a
```

The most interesting instances are for the meta-information of a constructor and a selector. For simplicity, we always place parentheses around a constructor and ignore infix operators. We do display a labeled constructor with record notation. At the constructor level, we use *conIsRecord* to decide if we print surrounding brackets or not. We use the *Bool* argument to $shows_1$ to encode that we are inside a labeled field, as we will need this for the product case:

```
instance (Show1 φ, Constructor γ) ⇒ Show1 (M1 C γ φ) where
  shows1 _ c@(M1 a) =
      showString "(" ∘ showString (conName c)
    ∘ showString " "
    ∘ wrapRecord
        (shows1 (conIsRecord c) a ∘ showString ")")
    where
      wrapRecord :: ShowS → ShowS
      wrapRecord s | conIsRecord c =  showString "{ " ∘ s
                                   ∘ showString " }"
      wrapRecord s | otherwise     = s
```

For a selector, we print its label (as long as it is not empty), followed by an `"="` and the value. In the product, we use the *Bool* to decide if we print a space (unlabeled constructors) or a comma:

```
instance (Show1 φ, Selector γ) ⇒ Show1 (M1 S γ φ) where
  shows1 b s@(M1 a)
    | null (selName s) = shows1 b a
    | otherwise        =  showString (selName s)
                        ∘ showString " = " ∘ shows1 b a
instance (Show1 φ, Show1 ψ) ⇒ Show1 (φ × ψ) where
  shows1 b (a × c) =  shows1 b a
                   ∘ showString (if b then "," else " ")
                   ∘ shows1 b c
```

Finally, we provide the default:

```
{-# DERIVABLE Show shows showsDefault #-}
showsDefault :: (Representable0 α τ, Show1 τ)
             ⇒ τ χ → α → ShowS
showsDefault rep x = shows1 False (from0 x `asTypeOf` rep)
```

We have shown how to use meta-information to define a generic *show* function. If we additionally account for infix constructors and operator precedence for avoiding unnecessary parentheses, we obtain a formal specification of how *show* behaves on every Haskell 98 datatype.

## 4. Compiler support

We now describe in detail the required compiler support for our generic deriving mechanism.

We start by defining two predicates on types, $\mathsf{isRep}_0\ (\phi)$ and $\mathsf{isRep}_1\ (\phi)$, which hold if $\phi$ can be made an instance of $Representable_0$ and $Representable_1$, respectively. The statement $\mathsf{isRep}_0\ (\phi)$ holds if $\phi$ is any of the following:

1. A regular Haskell 98 datatype without context
2. An empty datatype
3. A type variable of kind $\star$

We also require that for every type $\psi$ that appears as an argument to a constructor of $\phi$, $\mathsf{isRep}_0\ (\psi)$ holds. $\phi$ cannot use existential quantification, type equalities or any other extensions.

The statement $\mathsf{isRep}_1\ (\phi)$ holds if the following conditions both hold:

1. $\mathsf{isRep}_0\ (\phi)$
2. $\phi$ is of kind $\star \to \star$ or $k \to \star \to \star$, for any kind $k$

Note that $\mathsf{isRep}_0$ holds for all the types of Section 2.4, while $\mathsf{isRep}_1$ holds for $List$, $Expr$, $Decl$, and $Var$.

Furthermore, we define the predicate $\mathsf{ground}\ (\phi)$ to determine whether or not a datatype has type variables. For instance, $\mathsf{ground}\ ([Int])$ holds, but $\mathsf{ground}\ ([\alpha])$ not. Finally, we assume the existence of an indexed fresh variable generator **fresh** $p_i^j$, which binds $p_i^j$ to a unique fresh variable.

For the remainder of this section, we consider a user-defined datatype

$$\begin{aligned}\textbf{data}\ D\ \alpha_1 \ldots \alpha_n = {} & Con_1\ \{l_1^1 :: p_1^1,\ \ldots, l_1^{o_1} :: p_1^{o_1}\} \\ & \vdots \\ \mid {} & Con_m\ \{l_m^1 :: p_m^1, \ldots, l_m^{o_m} :: p_m^{o_m}\}\end{aligned}$$

with $n$ type parameters, $m$ constructors and possibly labeled parameter $l_i^j$ of type $p_i^j$ at position $j$ of constructor $Con_i$.

### 4.1 Type representation (kind $\star$)

In Figure 1, we show how we generate type representations for a datatype $D$ satisfying $\mathsf{isRep}_0\ (D)$. We generate a number of empty datatypes which we use in the meta-information: one for the datatype, one for each constructor and one for each argument to a constructor.

The type representation is a type synonym ($Rep_0^D$) with as many type variables as $D$. It is a wrapped sum of wrapped products: the wrapping encodes the meta-information. We wrap all arguments to constructors, even if the constructor is not a record. Since we use a balanced sum (resp. product) encoding, a generic function can use the meta-information to find out when the sum (resp. product) structure ends, which is when we reach $C_1$ (resp. $S_1$). Each argument is tagged with $Par_0$ if it is one of the type variables, or $Rec_0$ if it is anything else (type application or a concrete datatype).

### 4.2 $Representable_0$ instance

The instance $Representable_0\ Rep_0^D$ is defined in Figure 2, as introduced in Section 2. The patterns of the $from_0$ function are the constructors of the datatype applied to fresh variables. The same patterns become expressions in function $to_0$. The patterns of $to_0$ are also the same as the expressions of $from_0$, and they represent the different values of a balanced sum of balanced products, properly wrapped to account for the meta-information. Note that, for $Representable_0$, the functions **tuple** and **wrap** do not behave differently depending on whether we are in $from_0$ or $to_0$, so for these declarations the *dir* argument is not needed. Similarly, the **wrap** function could have been inlined. These definitions will be refined in Section 4.4.

### 4.3 Type representation (kind $\star \to \star$)

See Figure 3 for the type representation of type constructors. We keep the sum-of-products structure and meta-information unchanged. At the arguments, however, we can use $Par_0$, $Par_1$, $Rec_0$, $Rec_1$, or composition. We use $Par_1$ for the type variable $\alpha$, and $Par_0$ for other type variables of kind $\star$. A recursive occurrence of a type containing $\alpha_n$ is marked with $Rec_1$. A recursive occurrence of a type with no type variables is marked with $Rec_0$, as there is no variable to abstract from. Finally, for a recursive occurrence of a type which contains something else than $\alpha_n$ we use composition, and recursively analyze the contained type.

### 4.4 $Representable_1$ instance

The definition of the embedding-projection pair for kind $\star \to \star$ datatypes, shown in Figure 4, reflects the more complicated type representation. The patterns are unchanged. However, the expressions in $to_1$ need some additional unwrapping. This is encoded in **var** and **unwC**: an application to a type variable other than $\alpha_n$ has been encoded as a composition, so we need to unwrap the elements of the contained type. We use *fmap* for this purpose: since we require $\mathsf{isRep}_1\ (\phi)$, we know that we can use *fmap* (see Section 3.4). The user should always derive $Functor$ for container types, as these can appear to the left of a composition.

Unwrapping is dual to wrapping: we use $Par_1$ for the type parameter $\alpha_n$, $Rec_1$ for containers of $\alpha_n$, $K_1$ for other type parameters and ground types, and composition for application to types other than $\alpha_n$. Considering composition, in $to_1$ we generate only $Comp_1$ applied to a fresh variable, as this is a pattern; the necessary unwrapping of the contained elements is performed in the right-hand side expression. In $from_1$ the contained elements are tagged properly: this is performed by $\mathbf{wC}\alpha$.

### 4.5 Meta-information

We generate three meta-information instances. For datatypes, we generate

$$\begin{aligned}&\textbf{instance}\ Datatype\ \$D\ \textbf{where} \\ &\quad moduleName\ \_ = mName \\ &\quad datatypeName\ \_ = dName\ ,\end{aligned}$$

where *dName* is a $String$ with the unqualified name of datatype $D$ and *mName* is a $String$ with the name of the module in which $D$ is defined.

For constructors, we generate

$$\begin{aligned}&\textbf{instance}\ Constructor\ \$Con_i\ \textbf{where} \\ &\quad conName\ \_ = name \\ &\quad \{conFixity\ \_ = fixity\} \\ &\quad \{conIsRecord\ \_ = True\}\ ,\end{aligned}$$

where $i \in 1..m$, and *name* is the unqualified name of constructor $Con_i$. The braces around *conFixity* indicate that this method is only defined if $Con_i$ is an infix constructor. In that case, *fixity* is $Infix$ *assoc prio*, where *prio* is an integer denoting the priority of $Con_i$, and *assoc* is one of $LeftAssociative$, $RightAssociative$, or $NotAssociative$. These are derived from the declaration of $Con_i$ as an infix constructor. The braces around *conIsRecord* indicate that this method is only defined if $Con_i$ uses record notation.

For all $i \in \{1..m\}$, we generate

$$\textbf{instance}\ Selector\ \$L_i^j\ \{\textbf{where}\ selName\ \_ = l_i^j\}\ ,$$

where $j \in \{1..o_i\}$. The brackets indicate that the instance is only given a body if $Con_i$ uses record notation. Otherwise, the default implementation for *selName* is used, i.e. *const* "".

$$\textbf{data}\ \$D$$
$$\textbf{data}\ \$Con_1$$
$$\vdots$$
$$\textbf{data}\ \$Con_m$$
$$\textbf{data}\ \$L_1^1$$
$$\vdots$$
$$\textbf{data}\ \$L_m^{o_m}$$

$$\textbf{type}\ Rep_0^D\ \alpha_1 \ldots \alpha_n = D_1\ \$D\ (\textstyle\sum_{i=1}^{m}\ (C_1\ \$Con_i\ (\prod_{j=1}^{o_m}\ (S_1\ \$L_i^j\ (\textbf{arg}\ p_i^j)))))$$

$$\begin{aligned}
\textstyle\sum_{i=1}^{n} x \mid n \equiv 0 &= V_1 \\
\mid n \equiv 1 &= x \\
\mid otherwise &= \textstyle\sum_{i=1}^{m} x + \sum_{i=1}^{n-m} x \ \ \textbf{where}\ m = \lfloor n/2 \rfloor \\
\textstyle\prod_{i=1}^{n} x \mid n \equiv 0 &= U_1 \\
\mid n \equiv 1 &= x \\
\mid otherwise &= \textstyle\prod_{i=1}^{m} x \times \prod_{i=1}^{n-m} x \ \textbf{where}\ m = \lfloor n/2 \rfloor
\end{aligned}$$

$$\begin{aligned}
\textbf{arg}\ p_i^j \mid \exists_{k \in \{1..n\}} : p_i^j \equiv \alpha_k &= Par_0\ p_i^j \\
\mid otherwise &= Rec_0\ p_i^j
\end{aligned}$$

**Figure 1.** Code generation for the type representation (kind $\star$)

$$\textbf{instance}\ Representable_0\ (D\ \alpha_1 \ldots \alpha_n)\ (Rep_0^D\ \alpha_1 \ldots \alpha_n)\ \textbf{where}\ \{$$

$$\begin{aligned}
from_0\ \textbf{pat}_1^{from} &= \textbf{exp}_1^{from}; & to_0\ \textbf{pat}_1^{to} &= \textbf{exp}_1^{to}; \\
&\vdots & &\vdots \\
from_0\ \textbf{pat}_m^{from} &= \textbf{exp}_m^{from}; & to_0\ \textbf{pat}_m^{to} &= \textbf{exp}_m^{to}; \}
\end{aligned}$$

$$\begin{aligned}
\textbf{exp}_i^{to} &= \textbf{pat}_i^{from} = Con_i\ (\textbf{fresh}\ p_i^1) \ldots (\textbf{fresh}\ p_i^{o_i}) \\
\textbf{exp}_i^{from} &= \textbf{pat}_i^{to} = M_1\ (\textbf{inj}_{i,m}\ (M_1\ (\textbf{tuple}_{\bar{i}}\ (p_i^1 \ldots p_i^{o_i}))))
\end{aligned}$$

$$\begin{aligned}
\textbf{inj}_{i,m}\ x \mid m \equiv 0 &= \bot \\
\mid m \equiv 1 &= x \\
\mid i \leqslant m' &= L_1\ (\textbf{inj}_{i,m'}\ x) \\
\mid i > m' &= R_1\ (\textbf{inj}_{i',m-m'}\ x) \\
\textbf{where}\ m' &= \lfloor m/2 \rfloor \\
i' &= \lfloor i/2 \rfloor
\end{aligned}$$

$$\begin{aligned}
\textbf{tuple}_i^{dir}\ (p_i^j \ldots p_i^{o_i}) \mid o_i \equiv 0 &= M_1\ U_1 \\
\mid o_i \equiv j &= M_1\ (\textbf{wrap}^{dir}\ (\textbf{fresh}\ p_i^j)) \\
\mid otherwise &= (\textbf{tuple}_i^{dir}\ (p_i^1 \ldots p_i^k)) \times (\textbf{tuple}_i^{dir}\ (p_i^{k+1} \ldots p_i^m)) \\
&\ \ \textbf{where}\ k = \lfloor o_i/2 \rfloor \\
\textbf{wrap}^{dir}\ p &= K_1\ p
\end{aligned}$$

**Figure 2.** Code generation for the $Representable_0$ instance

$$\textbf{type}\ Rep_1^D\ \alpha_1 \ldots \alpha_{n-1} = D_1\ \$D\ (\textstyle\sum_{i=1}^{m}\ (C_1\ \$Con_i\ (\prod_{j=1}^{o_m}\ (S_1\ \$L_i^j\ (\textbf{arg}\ p_i^j)))))$$

$$\begin{aligned}
\textbf{arg}\ p_i^j \mid \exists_{k \in \{1..n-1\}} : p_i^j \equiv \alpha_k &= Par_0\ p_i^j \\
\mid p_i^j \equiv \alpha_n &= Par_1 \\
\mid p_i^j \equiv \phi\ \alpha_n \wedge \mathsf{isRep}_1\ (\phi) &= Rec_1\ p_i^j \\
\mid p_i^j \equiv \phi\ \beta\ \ \wedge \mathsf{isRep}_1\ (\phi) \wedge \neg\ \mathsf{ground}\ (\beta) &= \phi \circ \textbf{arg}\ \beta \\
\mid otherwise &= Rec_0\ p_i^j
\end{aligned}$$

$\textstyle\sum_{i=1}^{m} x$ and $\prod_{j=1}^{n} x$ as in Figure 1.

**Figure 3.** Code generation for the type representation (kind $\star \to \star$)

$$\textbf{instance}\ Representable_1\ (D\ \alpha_1 \ldots \alpha_{n-1})\ (Rep_1^D\ \alpha_1 \ldots \alpha_{n-1})\ \textbf{where}\ \{$$

$$\begin{aligned}
from_1\ \textbf{pat}_1^{from} &= \textbf{exp}_1^{from}; & to_1\ \textbf{pat}_1^{to} &= \textbf{exp}_1^{to}; \\
&\vdots & &\vdots \\
from_1\ \textbf{pat}_m^{from} &= \textbf{exp}_m^{from}; & to_1\ \textbf{pat}_m^{to} &= \textbf{exp}_m^{to}; \}
\end{aligned}$$

$$\begin{aligned}
\textbf{exp}_i^{to} &= Con_i\ (\textbf{var}\ p_i^1) \ldots (\textbf{var}\ p_i^{o_i}) \\
\textbf{var}\ p_i^j \mid p_i^j \equiv \phi\ \alpha \wedge \alpha \not\equiv \alpha_n & \\
\wedge\ \mathsf{isRep}_1\ (\phi) &= fmap\ \textbf{unwC}_\alpha\ (\textbf{fresh}\ p_i^j) \\
\mid otherwise &= \textbf{fresh}\ p_i^j
\end{aligned}$$

$\textbf{pat}_i^{dir}$, $\textbf{exp}_i^{from}$, $\textbf{inj}_{i,m}\ x$, and $\textbf{tuple}_i^{dir}\ (p_1 \ldots p_m)$ as in Figure 2 (but using the new $\textbf{wrap}^{dir}\ x$).

$$\begin{aligned}
\textbf{wrap}^{dir}\ p_i^j \mid p_i^j \equiv \alpha_n &= Par_1\ (\textbf{fresh}\ p_i^j) \\
\mid p_i^j \equiv \phi\ \alpha_n \wedge \mathsf{isRep}_1\ (\phi) &= Rec_1\ (\textbf{fresh}\ p_i^j) \\
\mid \exists_{k \in \{1..n\}} : p_i^j \equiv \alpha_k &= K_1\ (\textbf{fresh}\ p_i^j) \\
\mid p_i^j \equiv \phi\ \alpha \wedge \neg\ \mathsf{isRep}_1\ (\phi) &= K_1\ (\textbf{fresh}\ p_i^j) \\
\mid p_i^j \equiv \phi\ \alpha \wedge dir \equiv from &= Comp_1\ (fmap\ \textbf{wC}_\alpha\ (\textbf{fresh}\ p_i^j)) \\
\mid otherwise &= Comp_1\ (\textbf{fresh}\ p_i^j)
\end{aligned}$$

$$\begin{aligned}
\textbf{unwC}_\alpha \mid \alpha \equiv \alpha_n &= unPar_1 \\
\mid \alpha \equiv \phi\ \alpha_n \wedge \mathsf{isRep}_1\ (\phi) &= unRec_1 \\
\mid \alpha \equiv \phi\ \beta\ \ \wedge \mathsf{ground}\ (\beta) &= unRec_0 \\
\mid \alpha \equiv \phi\ \beta\ \ \wedge \mathsf{isRep}_1\ (\phi) &= fmap\ \textbf{unwC}_\beta \circ unComp_1 \\
\textbf{wC}_\alpha \mid \alpha \equiv \alpha_n &= Par_1 \\
\mid \mathsf{ground}\ (\alpha) &= K_1 \\
\mid \alpha \equiv \phi\ \alpha_n \wedge \mathsf{isRep}_1\ (\phi) &= Rec_1 \\
\mid \alpha \equiv \phi\ \beta\ \ \wedge \mathsf{isRep}_1\ (\phi) &= Comp_1 \circ (fmap\ \textbf{wC}_\beta)
\end{aligned}$$

**Figure 4.** Code generation for the $Representable_1$ instance

### 4.6 Default instances

The instances of a class representing the different cases of a generic function on representation types present somewhat more of a challenge because they refer to a specific function defined by the generic programmer (in our running example *encodeDefault*). The compiler knows which function to use due to the `DEFAULT` pragma (Section 3.3).

After the default function has been determined, the only other concern is passing the explicit type representation, encoded as a typed $\perp$.

#### 4.6.1 Generic functions on $Representable_0$

For each generic function $f$ that is a method of the type class $F$, and for every datatype $D$ with type arguments $\alpha_1 \ldots \alpha_n$ and associated representation type $Rep_0^D\ \alpha_1 \ldots \alpha_n\ \chi$, the compiler generates:

**instance** $(C \ldots) \Rightarrow F\ (D\ \alpha_1 \ldots \alpha_n)$ **where**
  $f = f_D\ \perp$ **where**
    $f_D :: (C \ldots) \Rightarrow Rep_0^D\ \alpha_1 \ldots \alpha_n\ \chi \to \beta$
    $f_D = f_{Default}$

The type $\beta$ is the type of $f$ specialized to $D$, and $\chi$ is a fresh type variable. The context $C$ is the same in the instance head and in function $f_D$. The exact context generated depends on the way the user specified the deriving. If **deriving** $F$ was attached to the datatype, we generate a context $F\ \overrightarrow{\alpha_1}, \ldots, F\ \overrightarrow{\alpha_n}$, where $\overrightarrow{\alpha}$ is the variable $\alpha$ applied to enough fresh type variables to achieve full saturation. This approach gives the correct behavior for Haskell 98 derivable classes like *Show*. In general, however, it is not correct: we cannot assume that we require $F\ \alpha_i$ for all $i \in \{1..n\}$: generic *children*, for instance, does not require any constraints, as it is not a recursive function. Worse even, we might require constraints other than these, as a generic function can use other functions, for instance.

To avoid these problems we can use the standalone deriving extension. If we have a standalone deriving

**deriving instance** $(C \ldots) \Rightarrow F\ (D\ \alpha_1 \ldots \alpha_n)$

we can simply use this context for the instance. In general, however, the compiler should be able to infer the right context by analyzing the context of the generic function and the structure of the datatype.

#### 4.6.2 Generic functions on $Representable_1$

For each generic function $f$ that is a method of the type class $F$, and for every datatype $D$ with type arguments $\alpha_1 \ldots \alpha_n$ and associated representation type $Rep_1^D\ \alpha_1 \ldots \alpha_n$, the compiler generates:

**instance** $(C \ldots) \Rightarrow F\ (D\ \alpha_1 \ldots \alpha_{n-1})$ **where**
  $f = f_D\ \perp$ **where**
    $f_D :: (C \ldots) \Rightarrow Rep_1^D\ \alpha_1 \ldots \alpha_n \to \beta$
    $f_D = f_{Default}$

The type $\beta$ is the type of $f$ specialized to $D$ (in other words, $f :: \beta$). This code is almost the same as that for generic functions on $Representable_0$, with a small exception for handling the last type variable ($\alpha_n$). The context can be copied from the standalone deriving, if one was used, or just inferred by the compiler.

### 4.7 UHC specifics

We have a prototype implementation of our deriving mechanism in UHC. Although generating the required datatypes and instances is straightforward, we have to resolve some subtle issues. In our implementation, the following issues arose:

***Which stage of the compiler pipeline generates the datatypes and instances?*** Ideally, all deriving-related code is generated as early as possible, for example during desugaring, so later compiler stages can type check the generated code. However, the generation needs kind information of types and classes, which is only available after kind checking. In UHC, the datatypes and instances are directly generated as intermediate Core, directed by kind information, and only the derived instances are intertwined with type checking and context reduction because of the use of the default deriving functions.

***Use of fmap.*** The generation of embedding-projection pairs for types with composition requires *fmap*, which in turn requires the context reduction machinery to resolve overloading. This complicates the interaction with the compiler pipeline, because the generation becomes not only kind-directed, but also context reduction proof-directed. However, all occurrences of *fmap* are applied to the identity function *id*, because wrappers like $Par_1$ are defined as newtypes. In UHC, the use of context reduction is avoided assuming the equality $fmap\ id \equiv id$.

***Code size.*** Some quick measurements show a 10% increase in the size of the generated code. Although language pragmas like *GenericDeriving* and *NoGenericDeriving* could selectively switch this feature on or off, this would defeat the purpose of genericity. Once turned off for a datatype, no *Representable*s are generated, and no generic instances can be defined anymore. Instead, later transformations should prune unused code. These issues need further investigation.

***Bootstrapping.*** As soon as a user defines a datatype, code generation generates the supporting datatypes. Such datatypes (e.g. $\$Con_1$) and the datatypes used by supporting datatypes (e.g. *Bool*, used in the return type of *conIsRecord*) are mutually dependent, which is detected by binding group analysis. Each binding group type analysis must deal with mutually dependent datatypes. This also means that the supporting definitions must be available in the first module that contains a datatype.

***Interaction with desugaring.*** Currently, **deriving** clauses are just syntactic sugar for standalone deriving. After desugaring, we cannot decide to generate a $Representable_0$ or a $Representable_1$ instance because kind information is not available. Automatically generating the correct context for such an instance cannot be done either. To work around this limitation, we only accept **deriving** clauses for generic classes that use $Representable_0$. Derivings for $Representable_1$ classes have to use standalone deriving syntax, since then we no longer need to infer a context, and can let the programmer provide the required context.

## 5. Alternatives

We have described how to implement a **deriving** mechanism that can be used to specify many datatype-generic functions in Haskell. There are other alternatives, of varying complexity and type-safety.

### 5.1 Pre-processors

The simplest, most powerful and least type safe alternative to our approach is to implement **deriving** by pre-processing the source file(s), analyzing the datatypes definitions and generating the required instances with a tool such as DrIFT (Winstanley and Meacham 2008). This requires no work from the compiler writer, but does not simplify the task of adding new derivable classes, as programming by generating strings is not very convenient.

Staged meta-programming lies in between a pre-processor and an embedded datatype-generic representation. GHC supports Template Haskell (Sheard and Peyton Jones 2002), which has become a standard tool for obtaining reflection in Haskell. While Template Haskell provides possibly more flexibility than the purely library-based approach we describe, it imposes a significant hurdle on the

compiler writer, who does not only have to implement a language for staged programming (if one does not yet exist for the compiler, like in UHC), but also keep this complex component up-to-date with the rest of the compiler, as it evolves. As an example, Template Haskell support for GADTs and type families only arrived much later than the features themselves. Also, for the derivable class writer, using Template Haskell is more cumbersome and error-prone than writing a datatype-generic definition in Haskell itself.

For these reasons we think that our library-based approach, while having some limitations, has a good balance of expressive power, type safety, and the amount of implementation work required.

### 5.2 Generic programming libraries

Another design choice we made was in the specific library approach to use. We have decided not to use any of the existing libraries but instead to develop yet another one. However, our library is merely a variant of existing libraries, from which it borrows many ideas. We see our representation as a mixture between `regular` (Van Noort et al. 2008) and `instant-generics` (Chakravarty et al. 2009). We share the functorial view with `regular`; however, we abstract from a single type parameter, and not from the recursive occurrence. Our library can also be seen as `instant-generics` extended with a single type parameter. However, having one parameter allows us to deal with composition effectively, and we do not duplicate the representation for types without parameters.

Since we wanted to avoid using GADTs, and we wanted an extensible approach, we had to exclude most of the other generic programming libraries. The only possible choice would have been `EMGM` (Oliveira et al. 2007), which supports type parameters, is modular and does not require fancy extensions. However, `EMGM` duplicates the representation for higher arities, and encodes the representation of a type at the value level. We prefer encoding the representation only at the type level, as this has proven to allow for type-indexed datatypes (see Section 7.2).

## 6. Related work

The generic programming library we present shares many aspects with `regular` (Van Noort et al. 2008) and `instant-generics` (Chakravarty et al. 2009). Clean (Alimarine and Plasmeijer 2001) has also integrated generic programming directly in the language. We think our approach is more lightweight: we express our generic functions almost entirely in Haskell and require only one small syntactic extension. On the other hand, the approach taken in Clean allows defining generic functions with polykinded types (Hinze 2002), which means that the function *bimap* (see Section 2.1), for instance, can be defined. Not all Clean datatypes are supported: quantified types, for example, cannot derive generic functions. Our approach does not support all features of Haskell datatypes, but most common datatypes and generic functions are supported.

An extension for derivable type classes similar to ours has been developed by Hinze and Peyton Jones (2001) in GHC. As in Clean, this extension requires special syntax for defining generic functions, which makes it harder to implement and maintain. In contrast, generic functions written in our approach are portable across different compilers. Furthermore, Hinze and Peyton Jones's approach cannot express functions such as *fmap*, as their type representation does not abstract over type variables.

Rodriguez Yakushev et al. (2008) give criteria for comparing generic programming libraries. These criteria consider the library's use of types, and its expressiveness and usability. Regarding types, our library scores very good: we can represent regular, higher-kinded, nested, and mutually recursive datatypes. We can also express subuniverses: generic functions are only applicable to types that derive the corresponding class. We only miss the ability to represent nested higher-kinded datatypes, as our representation abstracts only over a parameter of kind $\star$.

Regarding expressiveness, our library scores good for most criteria: we can abstract over type constructors, give ad-hoc definitions for datatypes, our approach is extensible, supports multiple generic arguments, represents the constructor names and can express consumers, transformers, and producers. We cannot express *gmapQ* in our approach, but our generic functions are still first-class: we can call generic map with generic show as argument, for instance. Ad-hoc definitions for constructors would be of the form:

$$\begin{aligned}&\textbf{instance } \mathit{Show}\ \mathit{Exp} \textbf{ where}\\ &\quad \mathit{shows}\ (\mathit{Plus}\ e_1\ e_2) = \mathit{shows}\ e_1 \circ \mathit{showString}\ \texttt{"+"} \circ \mathit{shows}\ e_2\\ &\quad \mathit{shows}\ x \qquad\qquad = \mathit{shows}_{\mathit{Default}}\ (\bot :: \mathit{Rep}_0^{\mathit{Exp}}\ \chi)\ x\end{aligned}$$

However, in our current implementation, $\mathit{Rep}_0^{\mathit{Exp}}$ is an internal type synonym not exposed to the user. Exposing it to the user would require a naming convention. If UHC supported type families (Schrijvers et al. 2008), $\mathit{Rep}_0$ could be a visible type family, which would solve our problem for ad-hoc definitions of constructors. It would also remove the need for using *asTypeOf* in Section 2.3.

Regarding usability, our approach supports separate compilation, is highly portable, has automatic generation of its two representations, requires minimal work to instantiate and define a generic function, is implemented in a compiler and is easy to use. We have not yet benchmarked our library in UHC. In GHC, we believe it will be as efficient as `instant-generics` and `regular`.

## 7. Future work

Our solution is applicable to a wide range of datatypes and can express many generic functions. However, some limitations still remain, and many improvements are possible. In this section we outline some possible directions for future research.

### 7.1 Supported datatypes

Our examples in Section 2 show that we can represent many common forms of datatypes. We believe that we can represent all of the Haskell 98 standard datatypes in $\mathit{Representable}_0$, except for constrained datatypes. We could easily support constrained datatypes by propagating the constraints to the generic instances.

Regarding $\mathit{Representable}_1$, we can represent many, but not all datatypes. Consider a nested datatype for representing balanced trees:

$$\textbf{data}\ \mathit{Perfect}\ \rho = \mathit{Node}\ \rho \mid \mathit{Perfect}\ (\rho, \rho)$$

We cannot give a representation of kind $\star \to \star$ for *Perfect*, since for the *Perfect* constructor we would need something like $\mathit{Perfect} \circ \mathit{Rec}_1\ ((,)\ \rho)$. However, the type variable $\rho$ is no longer available, because we abstract from it. This limitation is caused by the fact that we abstract over a single type parameter. The approach taken by Hesselink (2009) is more general and fits closely with our approach, but it is not clear if it is feasible without advanced language extensions.

Note that for this particular case we could use a datatype which pairs elements of a single type:

$$\textbf{data}\ \mathit{Pair}\ \rho = \mathit{Pair}\ \rho\ \rho$$

The representation for the *Perfect* constructor could then be $\mathit{Perfect} \circ \mathit{Rec}_1\ \mathit{Pair}$.

### 7.2 Type-indexed datatypes

Some generic functionality, like the zipper (Huet 1997) and generic rewriting (Van Noort et al. 2008), require not only type-indexed functions but also type-indexed datatypes: types that depend on the

structure of other types (Hinze et al. 2002). We plan to investigate how type-indexed datatypes can be integrated easily in our generic deriving mechanism, while still avoiding advanced language extensions.

### 7.3 Generic functions

The representation types we propose limit the kind of generic functions we can define. We can express the Haskell 98 standard derivable classes *Eq*, *Ord*, *Enum*, *Bounded*, *Show*, and *Read*, even lifting some of the restrictions imposed on the *Enum* and *Bounded* instances. All of these are expressible for $Representable_0$ types. Using $Representable_1$, we can implement *Functor*, as the parameter of the *Functor* class is of kind $\star \to \star$. The same holds for *Foldable* and *Traversable*. For *Typeable* we can express $Typeable_0$ and $Typeable_1$.

On the other hand, the *Data* class has very complex generic functions which cannot be expressed with our representation. Function *gfoldl*, for instance, requires access to the original datatype constructor, something we cannot do with the current representation. In the future we plan to explore if and how we can change our representation to allow us to express more generic functions.

### 7.4 Efficiency

The instances derived in our approach are not specialized for a datatype and may therefore incur an unacceptable performance penalty. However, our recent research (Magalhães et al. 2010) indicates that simple inlining and symbolic evaluation, present in some form in every optimizing compiler, suffice in most cases to optimize away all overhead from generic representations. We plan to investigate how these optimizations can be expressed and automatically applied without any user intervention in UHC.

### 7.5 Implementation in GHC

Our approach is designed to be as portable as possible. Therefore, we would like to implement it in other compilers, most importantly in GHC. As a first step, we believe we can easily implement most of our generic deriving mechanism in GHC using Template Haskell. The code for the generic functions is kept intact: only the `DERIVABLE` pragma needs a different syntax. For the user code, a code splice would trigger the generation of generic representations and function instances.

## 8. Conclusion

We have shown how datatype-generic programming can be better integrated in Haskell by revisiting the **deriving** mechanism. All Haskell 98 derivable type classes can be expressed as generic functions in our library, with the advantage of becoming easily readable and portable. Additionally, many other type classes, such as *Functor* and *Typeable*, can be declared derivable. Our extension requires little extra syntax, so it is easy to implement. Adding new generic derivings can be done by generic programmers in regular Haskell; previously, this would be the compiler developer's task, and would be done using code generation, which is more error-prone and verbose.

We have implemented our solution in UHC and invite everyone to derive instances for their favorite datatypes or even write their own derivings. We hope our work paves the future for a redefinition of the behavior of derived instances for Haskell Prime (Wallace et al. 2007).

## Acknowledgments

This work has been partially funded by the Portuguese Foundation for Science and Technology (FCT) via the SFRH/BD/35999/2007 grant. We thank Thomas van Noort and the anonymous reviewers for their helpful feedback.

## References

Artem Alimarine and Rinus Plasmeijer. A Generic Programming Extension for Clean. In *IFL'01*, pages 168–185. Springer-Verlag, 2001.

Roland Backhouse, Patrik Jansson, Johan Jeuring, and Lambert Meertens. Generic programming—an introduction. In *AFP'98*, volume 1608 of *LNCS*, pages 28–115. Springer, 1999.

Manuel M. T. Chakravarty, Gabriel C. Ditu, and Roman Leshchinskiy. Instant generics: Fast and easy, 2009. Draft version.

Atze Dijkstra, Jeroen Fokker, and S. Doaitse Swierstra. The architecture of the Utrecht Haskell compiler. In *Haskell'09*, pages 93–104. ACM, 2009.

Erik Hesselink. Generic programming with fixed points for parametrized datatypes. Master's thesis, Utrecht University, 2009.

Ralf Hinze. Polytypic values possess polykinded types. *SCP*, 43(2-3):129–159, 2002.

Ralf Hinze and Andres Löh. Generic programming in 3D. *SCP*, 74(8): 590–628, 2009.

Ralf Hinze and Simon Peyton Jones. Derivable type classes. *Electronic Notes in Theoretical Computer Science*, 41(1):5–35, 2001.

Ralf Hinze, Johan Jeuring, and Andres Löh. Type-indexed data types. In *MPC'02*, volume 2386 of *LNCS*, pages 148–174. Springer, 2002.

Ralf Hinze, Johan Jeuring, and Andres Löh. Comparing approches to generic programming in Haskell. In *Datatype-Generic Programming*, volume 4719 of *LNCS*, pages 72–149. Springer, 2007.

Gérard Huet. The zipper. *JFP*, 7(5):549–554, 1997.

Patrik Jansson and Johan Jeuring. PolyP—a polytypic programming language extension. In *POPL'97*, pages 470–482. ACM, 1997.

Mark Jones. Type classes with functional dependencies. In *ESOP'00*, volume 1782 of *LNCS*, pages 230–244. Springer, 2000.

Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate: a practical approach to generic programming. In *TLDI'03*, pages 26–37, 2003.

Ralf Lämmel and Simon Peyton Jones. Scrap more boilerplate: reflection, zips, and generalised casts. In *ICFP'04*, pages 244–255. ACM, 2004.

Andres Löh. *Exploring Generic Haskell*. PhD thesis, Utrecht University, 2004.

José Pedro Magalhães, Stefan Holdermans, Johan Jeuring, and Andres Löh. Optimizing generics is easy! In *PEPM'10*, pages 33–42. ACM, 2010.

Thomas van Noort, Alexey Rodriguez Yakushev, Stefan Holdermans, Johan Jeuring, and Bastiaan Heeren. A lightweight approach to datatype-generic rewriting. In *WGP'08*, pages 13–24. ACM, 2008.

Bruno C.d.S. Oliveira, Ralf Hinze, and Andres Löh. Extensible and modular generics for the masses. In *TFP'06*, pages 199–216. Intellect, 2007.

Simon Peyton Jones et al. *Haskell 98, Language and Libraries. The Revised Report*. Cambridge University Press, 2003. A special issue of JFP.

Alexey Rodriguez Yakushev, Johan Jeuring, Patrik Jansson, Alex Gerdes, Oleg Kiselyov, and Bruno C.d.S. Oliveira. Comparing libraries for generic programming in Haskell. In *Haskell'08*, pages 111–122. ACM, 2008.

Tom Schrijvers, Simon Peyton Jones, Manuel M. T. Chakravarty, and Martin Sulzmann. Type checking with open type functions. In *ICFP'08*, pages 51–62. ACM, 2008.

Tim Sheard and Simon Peyton Jones. Template metaprogramming for Haskell. In *Haskell'02*, pages 1–16. ACM, 2002.

Malcom Wallace et al. Derived instances—Haskell Prime. `http://hackage.haskell.org/trac/haskell-prime/wiki/DerivedInstances`, April 2007. [Online; accessed 07-June-2010].

Noel Winstanley and John Meacham. DrIFT user guide. `http://repetae.net/computer/haskell/DrIFT/drift.html`, February 2008. [Online; accessed 07-June-2010].