September 2006

RepLib: A library for derivable type classes

Stephanie Weirich

University of Pennsylvania, sweirich@cis.upenn.edu

Follow this and additional works at: http://repository.upenn.edu/cis_papers

Recommended Citation

Publisher URL: http://doi.acm.org/10.1145/1159842.1159844

This paper is posted at ScholarlyCommons. http://repository.upenn.edu/cis_papers/362
For more information, please contact repository@pobox.upenn.edu.
RepLib: A library for derivable type classes

Abstract
Some type class instances can be automatically derived from the structure of types. As a result, the Haskell language includes the deriving mechanism to automatic generates such instances for a small number of built-in type classes. In this paper, we present RepLib, a GHC library that enables a similar mechanism for arbitrary type classes. Users of RepLib can define the relationship between the structure of a datatype and the associated instance declaration by a normal Haskell functions that pattern-matches a representation types. Furthermore, operations defined in this manner are extensible-instances for specific types not defined by type structure may also be incorporated. Finally, this library also supports the definition of operations defined by parameterized types.

Keywords
generic programming, representation type, Haskell, type class

Comments
Publisher URL: http://doi.acm.org/10.1145/1159842.1159844
Abstract
Some type class instances can be automatically derived from the structure of types. As a result, the Haskell language includes the “deriving” mechanism to automatic generates such instances for a small number of built-in type classes. In this paper, we present RepLib, a GHC library that enables a similar mechanism for arbitrary type classes. Users of RepLib can define the relationship between the structure of a datatype and the associated instance declaration by a normal Haskell functions that pattern-matches a representation type. Furthermore, operations defined in this manner are extensible—instances for specific types not defined by type structure may also be incorporated. Finally, this library also supports the definition of operations defined by parameterized types.

Categories and Subject Descriptors D.1.1 [Programming Techniques]: Applicable (Functional) Programming

General Terms Design, Languages

Keywords Type-indexed programming, Datatype-generic programming, Representation types, GADT

1. Deriving type-indexed operations
Type-indexed functions are those whose behavior is determined by the types of their arguments. In Haskell, type classes [32, 8] enable the definition and use of such functions. For example, the $\textit{Eq}$ type class defines the signature of polymorphic equality.

\textbf{class} $\textit{Eq}\ a\ where\ (\equiv) : a \rightarrow a \rightarrow \textit{Bool}$

The instances of the $\textit{Eq}$ class define the behavior of polymorphic equality at specific types. For example, an instance for a datatype $\textit{Tree}$ is below.

\textbf{data} $\textit{Tree}\ a = \text{Leaf } a | \text{Branch } (\textit{Tree } a) (\textit{Tree } a)$

\textbf{instance} $\textit{Eq}\ a \Rightarrow \textit{Eq}\ (\textit{Tree } a)$\ where

\begin{align*}
(\text{Leaf } x1) \equiv (\text{Leaf } x2) &= x1 \equiv x2 \\
(\text{Branch } t1 \ t2) \equiv (\text{Branch } s1 \ s2) &= t1 \equiv s1 \land t2 \equiv s2 \\
- \equiv - &= \text{False}
\end{align*}

In general, when a programmer defines a new type $T$ in Haskell, she may enable polymorphic equality for that type by providing an instance of $\textit{Eq}\ T$.

However, Haskell programs often include many datatype definitions and it can be tiresome to define instances of $\textit{Eq}$ for all of these types. Furthermore, there is often a relationship between the structure of a datatype definition and its instance for $\textit{Eq}$, so many of these instances have similar definitions. As a result, the Haskell language includes the deriving mechanism that can be used to directly a Haskell compiler to insert an instance of the $\textit{Eq}$ based on the structure of a newly defined datatype. For example, the code above may be replaced by the following.

\textbf{data} $\textit{Tree}\ a = \text{Leaf } a | \text{Branch } (\textit{Tree } a) (\textit{Tree } a)$

\textbf{deriving} ($\textit{Eq}$)

Deriving is a useful addition to the Haskell language in that it cuts down on the boilerplate instance declarations that programmers must write when they declare new datatypes. Importantly, it is an optional mechanism, providing a default instance for $\textit{Eq}$ when directed, but allowing programmers to write their own specialized instances for $\textit{Eq}$ when necessary.

Unfortunately, deriving only works for a handful of built-in type classes. In Haskell 98, only $\textit{Eq}$, $\textit{Ord}$, $\textit{Bounded}$, $\textit{Show}$ and $\textit{Read}$ are derivable. User-defined type classes cannot take advantage of deriving. To address this limitation, there have been a number of proposals for experimental libraries and extensions to Haskell, such as Polypadic Programming (PolyP) [18], Generic Haskell [3, 24], Derivable type classes [11], the $\textit{Typeable}$ type class (with the “Scrap your Boilerplate Library” [21, 22, 23]), preprocessors such as DrIFT [6] and Template Haskell [30], and various encodings of representation types [39, 5, 13]. These proposals each have their benefits, but none has emerged as a clearly better solution.

In this paper, we present the RepLib library for the Glasgow Haskell Compiler (GHC) [7] that enables a deriving-like behavior for arbitrary type classes. It works by using Template Haskell to define representation types that programmers may use to specify the default behavior of type-indexed operations. Representation types reflect the structure of types as Haskell data, therefore programmers can define type-indexed operations as ordinary Haskell functions.

The idea of programming with representation types is itself not new. The contribution of this paper is instead four ideas that make it work in this particular situation. Individually, these ideas may seem small, but each is essential to the design. In short, the four ideas of this paper are:

- To make type classes “derivable” by using representation types to define default methods for them (Section 2).
- To generically represent the structure of datatypes with a list of data constructor embeddings (Section 3).
- To support specializable type-indexed operations by parameterizing the representation of datatypes with explicit dictionaries (Section 4).
- To support the definition of functions indexed by parameterized types by dynamically supplying explicit dictionaries (Section 5).
In Section 7, we compare the capabilities of this proposal to existing work. For example, there are a number of ways to generically represent the structure of datatypes, and, more broadly, there are a number of ways to define type-indexed operations that do not rely on representation types. However, our view is that the success of any proposal relies on ease of adoption. Therefore, we have worked hard to identify a small set of mechanisms, implementable within the language of an existing Haskell compiler, that are, in our subjective view, useful for common situations and provide a programming model familiar to functional programmers.

An initial release of RepLib is available for download\(^1\) and is compilable with the Glasgow Haskell Compiler (GHC), version 6.4. This library is not portable. It requires many of the advanced features of GHC that are not found in Haskell 98: Higher-rank polymorphism [29], lexically-scope type variables [31], Generalized Algebraic Datatypes (GADTs) [28], and undecidable instance declarations. Furthermore, Template Haskell [30] automates the definition of representations for new datatypes. However, all of these extensions are useful in their own respect.

2. Representation Types and Type Classes

We begin by showing how a simple representation type can be used to define a default method for a particular type class. The purpose of this Section is only to introduce representation types and clarify the roles that they and type classes play. The code developed here is for illustrative purposes and not part of the RepLib library.

Representation types [4] allow programmers to define type-indexed operations as they would many other functions in Haskell—by pattern matching an algebraic datatype. However, a representation type is no ordinary datatype: It is an example of a Generalized Algebraic Datatype (GADT), a recent addition to GHC [28].

For example, we define the representation type \(R\) below, following GADT notation, by listing all of its data constructors with their types.

\[
data R a where
\begin{align*}
    
    \text{Int} & :\!:: R \text{Int} \\
    \text{Unit} & :\!:: R () \\
    \text{Bool} & :\!:: R \text{Bool} \\
    \text{Char} & :\!:: R \text{Char} \\
    \text{Pair} & :\!:: R a \rightarrow R b \rightarrow R (a, b) \\
    \text{Arrow} & :\!:: R a \rightarrow R b \rightarrow R (a \rightarrow b) \\
    \text{List} & :\!:: R a \rightarrow R [a]
\end{align*}
\]

The important feature of the \(R\) type is that, even though it is a parameterized datatype, the data constructor determines the type parameter. For example, the data constructor \(\text{Int}\) requires that the type parameter be \(\text{Int}\). This reasoning works in reverse, too. If we know that the type of a term is \(\text{Int}\), then we know that the term must either be the data constructor \(\text{Int}\) or \(\_\_\_\)...

GHC performs this sort of reasoning when type checking type-indexed functions. For example, we might write an operation that adds together all of the \(\text{Ints}\) that appear in a data structure. (In this paper, all functions whose first argument is a representation type end with a capital “\(R\))

\[
gsumR :: R a \rightarrow a \rightarrow \text{Int} \\
gsumR \text{Int} x = x \\
gsumR (\text{Pair} t1 t2) (x1, x2) = gsumR t1 x1 + gsumR t2 x2 \\
gsumR (\text{List} t) l = foldl (\lambda s x \rightarrow (gsumR t x) + s) 0 l \\
gsumR (\text{Arrow} t1 t2) f = \text{error} \ "urk!" \\
gsumR \_ x = 0
\]

Operationally, this function is the identity function for integers. For compound data structures, such as lists and products, it decomposes its argument and calls itself recursively. Because we cannot access the integers that appear in a closure, it is an error to apply this function to data structures that contains functions. For all other types of arguments, this function returns 0.

This definition type checks in the \(\text{Int}\) branch because we know that in that branch the type \(a\) must be \(\text{Int}\). So, even though the type signature says the branch should return an \(\text{Int}\), it is acceptable to return the argument \(x\) of type \(a\). In GADT terminology, the type \(a\) has been refined to \(\text{Int}\). Furthermore, in the \(\text{Pair}\) branch, we know that the type \(a\) must be a tuple, so we may immediately destruct the argument. Likewise, in the \(\text{List}\) branch, \(l\) must be a list and so is an appropriate argument for foldl.

The \(gsizeR\) function may be applied to any argument composed of \(\text{Ints}\), unit, booleans, characters, pairs and lists, when provided with the appropriate type representation for that argument. For example,

\[
gsumR (\text{Bool} \text{Pair} (\text{List} \text{Int})) (\text{True}, [3, 4]) \equiv 7
\]

Now compare the definition of \(gsumR\) with a type-class based implementation. We could rewrite the generic sum function using type classes as:

\[
class GSum a where \\
gsum :: a \rightarrow \text{Int} \\
instance GSum \text{Int} where \\
gsum x = x \\
instance GSum () where \\
gsum x = 0 \\
instance GSum \text{Bool} where \\
gsum x = 0 \\
instance GSum \text{Char} where \\
gsum x = 0 \\
instance (GSum a, GSum b) \Rightarrow GSum (a, b) where \\
gsum (x1, x2) = gsum x1 + gsum x2 \\
instance GSum a \Rightarrow GSum [a] where \\
gsum l = foldl (\lambda s x \rightarrow (gsum x) + s) 0 l
\]

With this definition, only a little type information is required at the function call to disambiguate the \(\text{Num}\) class.

\[
gsum (\text{True}, [3, 4 :: \text{Int}]) \equiv 7
\]

Defining generic sum with type classes loses the simple notation of pattern matching (including the wildcard case) but has three significant advantages over the representation-based definition: easier invocation as seen above, a static description of the domain of \(gsum\), and extensibility to new types. By defining \(gsum\) with a type class we can statically prevent \(gsum\) from being called with types that contain functions, and we can extend the definition of \(gsum\) at any time with a case for a new user-defined type.

Disregarding the extensibility issue for the moment, we see that representation types make generic sum easier to define whereas type classes do it more easily to use. However, by using type classes and representation types together, we can get the advantages of both definitions.

Consider a class \(\text{Rep}\) that includes all types that are representable.

\[
class \text{Rep} a \ where \ rep :: R a
\]

The instances of this class are the data constructors of the representation type.

\[
\begin{align*}
    \text{instance Rep \text{Int} where} & \ rep = \text{Int} \\
    \text{instance Rep () where} & \ rep = \text{Unit} \\
    \text{instance Rep \text{Bool} where} & \ rep = \text{Bool}
\end{align*}
\]

\(^1\)http://www.cis.upenn.edu/~swierich/RepLib
instance Rep Char where rep = Char
instance (Rep a, Rep b) ⇒ Rep (a \,\to\, b)
   where rep = Pair rep rep
instance (Rep a, Rep b) ⇒ Rep (a \to b)
   where rep = Arrow rep rep
instance (Rep a) ⇒ Rep [a]
   where rep = List rep

We use this class by declaring that the class GSum is a subclass of Rep, which allows a default definition for the gsum method in terms of gsumR.

class Rep a ⇒ GSum a where
gsum :: a → Int
gsum = gsumR rep

Because of the default method, the instances of this class are trivial. In particular, there is no repeated logic between the instances and the definition of gsumR. Instead, the instances “derive” the definition of gsum for these particular types.

instance GSum Int
instance GSum ()
instance GSum Bool
instance GSum Char
instance (GSum a, GSum b) ⇒ GSum (a, b)
infixr 7 :::

Defining the type-indexed operation in this manner demonstrates the different roles that type classes and representation types should play. The representation-type implementation describes the behavior of the type-indexed operation and the type class limits its domain to acceptable types. Of course, the underlying implementation gsumR is still available, and the user must be careful not to call this operation with functions, but type classes make it more convenient to use gsum correctly.

However, we have gained little so far. The extensibility problem remains because this type class can only be instantiated for a handful of types. In the next section, we develop a more general representation type that can represent the structure of arbitrary datatypes and allow the definition of gsumR based on that structure.

3. Datatype-generic programming

The representation type defined in the previous section could only represent a handful of types. Furthermore, it does not allow us to implement gsumR based on the structure of the represented type. In particular, we would like to define the behavior of gsumR for both Pairs and Lists with the same code.

In this section, we describe a representation type that can generally represent the structure of all Haskell 98 datatypes. Consider the following revised definition of the R type:

data R a where
   Int :: R Int
   Char :: R Char
   Arrow :: R a → R b → R (a → b)
   Data :: DT → [Con R a] → R a

We represent all datatypes, both built-in and user-defined, with the new data constructor Data. Therefore, we no longer need the constructors List, Pair, Bool and Unit in the R type.

The Data constructor takes two arguments: information about the data type itself DT and information about each of the data constructors that make up the datatype (the list of Con R a). In Section 3.1 below, we begin our discussion with the design of Con and then in Section 3.2 we cover DT.

3.1 Representing data constructors

The Con datatype describes data constructors (such as Leaf or Branch).

data Con c a = ∀ l.Con (Emb l a) (MTup c l)

The parameter a is the datatype that these constructors belong to. The parameter c provides generality that will be used in the next section. Here, this parameter is always instantiated by the type R. This datatype includes three components, a type l that is a type list containing the types of the arguments of the constructor, an embedding-projection pair, Emb l a, between the arguments of the constructor and the datatype a, and MTup c l, the representation of the type list.

The ∀ in the definition of Con means that it includes an existential component [26]—an argument of type l is required for the data constructor Con, but l does not appear as an argument to the type constructor Con. Instead, l hides a type list (similar to a heterogeneous list [20]) so that we can uniformly represent data constructors that take different numbers and different types of arguments. Type lists are defined by the following two single-constructor datatypes. (By convention, the type variable a stands for an arbitrary type, while the type variable l stands for a type list.)

data Nil = Nil
data a := l = a ::: l
infixr 7 :::

Note that type lists generalize n-tuples. For example, the type (Int :: Char :: Nil) is isomorphic to the pair type (Int, Char).

example1 :: (Int :: Char :: Nil)
example1 = 2 :: 'b' :: Nil

The second ingredient we need in the representation of a data constructor for a datatype a is some way of manipulating arguments of type a in a generic way. In particular, given an a, we would like to be able to determine whether it is an instance of this particular data constructor, and if so extract its arguments. Also, given arguments of the appropriate types, we should be able to construct an a.

Therefore, Con includes an embedding-projection pair between the arguments of the constructor and the datatype, containing a generic version of a constructor and a generic destructor.

data Emb l a = Emb {to :: l → a, from :: a → Maybe l}

For example, below are the embedding-projection pairs for the constructors of the Tree datatype:

rLeafEmb :: Emb (a := Nil) (Tree a)
rLeafEmb = Emb
   {to = \(a := Nil\) → (Leaf a),
    from = \(a\) → case \(x\) of
           Leaf a → Just (a := Nil)
           _ → Nothing
   }

rLeafEmb = Emb
   {to = \(l := r := Nil\) → (Branch l r),
    from = \(ax\) → case \(x\) of
           Branch l r → Just (l := r := Nil)
           _ → Nothing
   }

Finally, the third component of the Con datatype is MTup c l, the representation of the type list l. We form this representation with the following GADT.
Like the \( R \) type, the type index describes what type list the term represents. The \((+:)\) constructor includes \( Rep \) in its context so that, as this list is destructed, this representation may be implicitly provided. For now, the \( c \) a component duplicates the representation in the context and is useful for disambiguation. In this way, the type \( MTup R l \) represents a list of types.

\[
example2 :: MTup R (Int :+: Char :+: Nil)
\]

To form the representations of the data constructors \( Leaf \) and \( Branch \), we need the representation of the type \( a \) to satisfy the class constraint of \((+:)\). The \( \forall \) in the type annotations of \( rLeaf \) and \( rBranch \) bind the lexically-scoped type variable \( a \) so that it may be used in the type annotations that specify which type representations to use.

\[
rLeaf :: \forall a. Rep a \Rightarrow Con (R (Tree a))
rLeaf = Con rLeafEmb (null :: R a) :+: MNil
\]

\[
rBranch :: \forall a. Rep a \Rightarrow Con (R (Tree a))
rBranch = Con rBranchEmb
\]

The definition of \( Con \) described in this section contains only the minimum information required for representing data constructors. In the the RepLib library implementation, this datatype also includes additional information about the data constructor, such as a string containing the name of the constructor, its fixity, and the names of any record labels. Here, we have elided those components.

### 3.2 The DT type

The \( DT \) component of the datatype representation contains information intrinsic to the datatype itself, including the name of the datatype and the representation of its parameters.

\[
data DT = \forall l. DT String (MTup R l)
\]

For example, we can represent the type \( Tree \) with the following instance of the \( Rep \) class.

\[
instance Rep a \Rightarrow Rep (Tree a) where
rep = Data (DT "Tree" ((null :: R a) :+: MNil)) [rLeaf, rBranch]
\]

Including the name of the datatype in its representation and the representations of any type parameters is necessary to distinguish between types that have the same structure. Therefore type-safe cast \([38]\) of type

\[
cast :: (Rep a, Rep b) \Rightarrow a \rightarrow Maybe b
\]

and the related generalized cast

\[
gcast :: (Rep a, Rep b) \Rightarrow c a \rightarrow Maybe (c b)
\]

can be implemented. Without this information, these operations cannot enforce the distinction between isomorphic type.\(^2\)

\(^2\)While the basic cast may be implemented by decomposing and reconstructing its argument, the implementation of the generalized cast requires the use of an unsafe type cast. However, for practical reasons, basic cast is also implemented with \(primUnsafeCoerce\#\) in the implementation.

### 3.3 Examples of type-indexed functions

Once we can represent datatypes structurally, we can define operations based on that structure. Consider the implementation of generic sum with this new representation:

\[
gsumR :: R a \rightarrow a \rightarrow Int
gsumR Int x = x
gsumR (Arrow r1 r2) f = error "urk"
gsumR (Data rdt cons) x = findCon cons
\]

\[
where
findCon (Con emb reps : rest) =
\]

\[
      case (from emb x) of
\]

\[
Just kids \rightarrow gsumRl reps kids
Nothing \rightarrow findCon rest
\]

\[
findCon [] = error "Invalid representation"
\]

Figure 1. Library operations for defining type-indexed functions of \( Show \) below displays a representation type. Note that pattern matching allows a natural definition for showing a list of type parameters.

\[
instance Show (R a) where
show Int = "%Int" show Char = "%Char" show (Arrow r1 r2) =
"% (show r1) + " \rightarrow " + (show r2) + ""
show (Data (DT str reps)) =
"% (str ++ show reps ++ ")"
\]

\[
instance Show (MTup R l) where
show MNil = ""
show (r :+: MNil) = show r
show (r :+: rs) = "c ++ show r ++ show rs
\]

In the case of \( Data \), the information about the data constructors is ignored. Instead the string and representations of the type parameters are used.

The representation of the datatype need only be created once, when the datatype is defined. (However, even if it is not done then, it may be created by any module that knows its definition.) In this way, \( Data \) may represent a wide range of datatypes, including parameterized datatypes (such as \( Tree \)), mutually recursive datatypes, nested datatypes, and some GADTs. Section 6.3 discusses the expressiveness of this representation type in more detail. Furthermore, given the definition of such datatypes (except for GADTs), RepLib includes Template Haskell code to automatically generate its representation and instance declaration for the \( Rep \) type class.
Like many other type-directed operations, deepSeq makes sense for all representable types. Therefore, we do not use a type class to govern its usage, only a wrapper to provide the representation argument from the context.

The operations gsum and deepSeq are examples of type-indexed consumers—functions that use type information to decompose an argument of the type. RepLib can also define producers. These functions, such as the zero operation below, create values of a given type.

```
class Rep a ⇒ Zero a where
g zero :: a ⇒ a
g zero = zeroR rep
zeroR :: R a ⇒ a
zeroR Int = 0
zeroR Char = '0'
zeroR (Arrow r1 r2) = const (zeroR r2)
zeroR (Data dt (Con emb rec : rest)) =
  to emb (fromTup zeroR rec)
  fromTup :: ∀ a. Rep a ⇒ c a → a → R c → l → l
  fromTup f MNil = Nil
  fromTup f (b :+: l) = (f b) :+: (fromTup f l)
```

"Scrap your boilerplate" programming

Representation types can implement many of the same operations as the "Scrap your boilerplate" (SYB) library by Lämmel and Peyton Jones [21]. For example, one part of the SYB library defines generic traversals over datatypes, using the type-indexed operations mkT, mapT and everywhere. Below, we show how to implement those operations with representation types.

A traversal is a function that has a specific behavior for a particular type (or set of types) but is the identity function everywhere else. In this setting, traversals have the following type:

```
type Traversal = ∀ a. Rep a ⇒ a → a
```

The mkT function constructs traversals by lifting a monomorphic function of type \( t \to t \) to be a Traversal.

```
mkT :: (Rep a, Rep b) ⇒ (a → a) → b → b
mkT f = case (cast f) of
  Just g → g
  Nothing → id
```

Next, the mapT function below extends a basic traversal to a "one-layer" traversal by mapping the traversal across the subcomponents of a data constructor. Note that the annotation of the return type \( a → a \) binds the lexically scoped type variable \( a \) so that we may refer to it in the annotation \( R a \).

```
mapT :: Traversal → Traversal
mapT t :: a → a =
case (rep :: R a) of
  (Data dt cons) → λx →
    case (findCon cons x) of
      Val emb kids →
        to emb (mapT (const t) kids) → id
```

Finally, the everywhere combinator applies the traversal to every node in a datatype. The definition of everywhere is exactly the same as in the SYB library.
everywhere :: Traversal → Traversal
\[
\text{everywhere } f = f (\text{mapT}(\text{everywhere } f))
\]

With these operations we can compile and execute the “paradise” benchmark. Although the definition of the type Traversal and the implementation of mapT are different in this setting, these operations may be used in exactly the same way as before. For example, an operation to increase all salaries in a Company data structure may be implemented with a single line, giving the interesting case for increasing salaries.

\[
\begin{align*}
\text{increase} & :: \text{Float} \rightarrow \text{Company} \rightarrow \text{Company} \\
\text{increase } k & = \text{everywhere } (\text{mkT}(\text{incS } k)) \\
\text{incS} & :: \text{Float} \rightarrow \text{Salary} \rightarrow \text{Salary} \\
\text{incS } k (s) & = s (s + (1 + k))
\end{align*}
\]

This implementation of SYB with representation types was inspired by toSpine view of datatypes of Hinze et al. [13]. The generic view in this paper is at least as expressive as that view—we could use it to implement their toSpine operation.

**Polymorphic equality** However, representation types are sometimes more natural to program with than the SYB library or spines. Both have difficulty with type-indexed producers, requiring new basic operations (such as gunfoldr) or a new view of types. Polymorphic equality is another example. It requires a “twin-traversal” scheme in SYB [22]. With spines, it must be generalized to comorphic equality is another example. It requires a “twin-traversal” scheme in SYB [22]. With spines, it must be generalized to comorphic equality is another example.

\[
\begin{align*}
\text{eqR} & :: R \rightarrow a \rightarrow a \rightarrow \text{Bool} \\
\text{eqR} & :: (\text{Arrow } t1 t2) \rightarrow \text{error } "urk" \\
\text{eqR} \text{ (Arrow cons)} & = \lambda x y \rightarrow \text{loop cons } x y \\
\text{where} & \text{ loop } (\text{Con emb reps } : \text{rest}) x y = \\
\text{case} (\text{from emb } x, \text{from emb } y) \text{ of} \\
(\text{Just } p1, \text{Just } p2) & \rightarrow \text{eqR } \text{reps } p1 p2 \\
(\text{Nothing}, \text{Nothing}) & \rightarrow \text{loop rest } x y \\
(\_ \_ ) & \rightarrow \text{False}
\end{align*}
\]

The above function determines how the structure of a type determines the implementation of polymorphic equality. However, the Eq class already exists as part of the Haskell Prelude, so we cannot modify it to use eqR as the default definition of (≡). However, for each specific type, we can use eqR in the Eq instance. For example, we may define polymorphic equality for trees with the following instance.

\[
\begin{align*}
\text{instance } (\text{Rep } a, \text{Eq } a) & \Rightarrow \text{Eq } (\text{Tree } a) \\
\text{where} & (\equiv ) = \text{eqR } \text{rep}
\end{align*}
\]

We might create such an instance when deriving it is not available—for example, if we could not modify the datatype declaration for Tree because it is another module. Note that, in this instance, we require that the parameter type a be a member of the Eq class even though we do not use the a definition of (≡). This constraint ensures that we do not call polymorphic equality on types, such as arrow types, that are representable but do not support polymorphic equality.

**Specializable type-indexed functions**

There is a serious problem with the definition of gsum presented in the previous section—it does not interact well with other instances of the GSum class. To make this issue more concrete, consider the following example. First, define a new type of sets of integers and its representation in the way described above.

\[
\begin{align*}
\text{newtype } & \text{IntSet } = IS [\text{Int}] \\
\text{rSEmb } & :: \text{Emb } ([\text{Int}] \Rightarrow \text{Nil}) \text{ IntSet} \\
\text{rSEmb } & = \text{Emb } \{ \text{to } = \lambda (d \Rightarrow \text{Nil}) \rightarrow IS \text{ d,} \\
& \quad \text{from } = \lambda (IS \text{ d}) \rightarrow \text{Just } (d \Rightarrow \text{Nil}) \}
\end{align*}
\]

\[
\begin{align*}
\text{instance } & \text{Rep } \text{IntSet} \text{ where} \\
\text{rep } & = \text{Data } (\text{DT } "\text{IntSet}" \text{ MNil}) \\
& \quad \text{[Con } \text{rSEmb } \{ \text{rep } :: R [\text{Int}] \Rightarrow \text{MNil} \}\]
\end{align*}
\]

Because sets are implemented as lists, there is no guarantee that the list will not contain duplicate elements. This means that we cannot use the default behavior of gsum for IntSet because these duplicate elements will be counted each time. Instead, we would like to use the following definition that first removes duplicates.

\[
\begin{align*}
\text{instance } & \text{GSum } \text{IntSet} \text{ where} \\
\text{gsum } & (IS \text{ l}) = \text{gsum } (\text{nub } \text{l})
\end{align*}
\]

Unfortunately, with this instance, the behavior of generic sum for IntSets depends on whether they appear at top level (where the correct definition is used) or within another data structure (where the default structure-based equality is used).

\[
\begin{align*}
\text{gsum } & (IS \text{ [1,1]}) \equiv 1 \\
\text{gsum } & (\text{Leaf } (IS \text{ [1,1]})) \equiv 2
\end{align*}
\]

To solve this problem, we introduce parameterized representations that allow type-indexed operations to be specialized for specific types.
4.1 Parameterized representations

The key idea for parameterized representations is to add a level of indirection. In a recursive call to a type-indexed function, we should first check to see if there is some specialized definition for that type instead of the generic definition. These recursive calls are made on the "kids" of data constructors. Concretely, we enable this check by augmenting the representations of data constructors with explicit dictionaries that possibly contain specific cases for a particular operation.

The dictionary may be for any type-indexed operation. Therefore, we parameterize the type \( R_1 \) below with the type of the dictionary. \( c \). A representation of type \( R_1 c a \) may only be used to define a type-indexed operation of type \( c a \). (Note that new definitions in this Section end with 1 to distinguish them from those of the previous section.)

```haskell
data R1 c a where
  Int1 :: R1 c Int
  Char1 :: R1 c Char
  Arrow1 :: (Rep a, Rep b) \Rightarrow c a \rightarrow c b \rightarrow R1 c (a \rightarrow b)
  Data1 :: DT \rightarrow [Con c a] \rightarrow R1 c a
```

As before, we create a (now multiparameter) type class to automatically supply type representations. So that we may continue to cast, we make this class a subclass of \( \text{Rep} \).

```haskell
class Rep a \Rightarrow Rep1 c a where
  rep1 :: R1 c a
```

A function to create representation types must abstract the contexts that should be supplied for each of the kids. For example, the representation of \( \text{Trees} \) below abstracts the explicit dictionaries \( ca \) and \( ct \) for the type parameters \( a \) and the type \( \text{Tree} a \) that appear in the kids of \( \text{Leaf} \) and \( \text{Branch} \).

```haskell
rTree1 :: \forall a c. (Rep a) \Rightarrow c a \rightarrow c (\text{Tree} a) \rightarrow R1 c (\text{Tree} a)
rTree1 ca ct =
  Data1 (DT "Tree" ((\text{rep} :: R a) \Rightarrow c a \rightarrow c (\text{Tree} a))
    [\text{Con rLeafEmb} (ca \Rightarrow :: MNil),
     \text{Con rBranchEmb} (ct \Rightarrow :: ct \Rightarrow :: MNil)]
```

It is the job of the the instance declaration that automatically creates the representation of the tree type to supply these dictionaries. These dictionaries are provided by instances of the type class \( \text{Sat} \). This type class can be thought of as a "singleton" type class, the class of types that contain a single value.\(^3\)

```haskell
class Sat a where
  dict :: a
```

The instance declaration for the representation of trees requires that the appropriate dictionaries be available. Note that this instance declaration requires undecidable instances as the constraint \( \text{Sat} (c \ (\text{Tree} a)) \) includes non-variables in the type.

```haskell
instance (Rep a, Sat (c a), Sat (c (\text{Tree} a))) \Rightarrow Rep1 c (\text{Tree} a) where
  rep1 = rTree1 dict dict
```

Likewise, the representation of \( \text{IntSet} \) requires an instance of \( \text{Sat} \) for its kid, of type \( [\text{Int}] \).

\(^3\)In fact, the type \( R \ t \) is also a singleton type for any \( t \). Therefore, we could replace the class \( \text{Rep} \) with \( \text{Sat} (R a) \).

```haskell
instance Sat (c [Int]) \Rightarrow Rep1 c IntSet where
  rep1 = Data1 (DT "IntSet" MNil)
    [\text{Con rSEmb} (dict \Rightarrow :: MNil)]
```

Creating parameterized representations is only half of the task. The other half is defining type-indexed operations so that they take advantage of this specializability. Consider the definition of a specializable version of generic sum, shown in Figure 3. The first step is to create a dictionary for this operation and a generic instance declaration for \( \text{Sat} \) for each type using this dictionary. This instance declaration stores whatever definition of polymorphic equality is available for the type \( a \) in the dictionary.

Next, we define the type-indexed operation with almost the same code as before. The only difference is the call \( gsumR1 \) that accesses the stored dictionary instead of calling \( gsumR1DT \) directly. In fact, we cannot call \( gsumR1DT \) recursively, as \( \text{Con} \) does not include \( R1 \) representations for its kids. This omission means that we must use the special cases for each type.

As a result, this time, the type-indexed definition of generic sum for trees uses the special case for \( \text{IntSets} \).

```haskell
isp (IS [1, 1]) \equiv 1
isp (Leaf (IS [1, 1])) \equiv 1
```

4.2 Calling other type-indexed operations

What if a type-indexed operation depends on other type-indexed operations? For example, a function to increase salaries may need to call an auxiliary function to determine whether the salary increase is eligible. One might think that this operation may be difficult to define here, as the parameterized representation type must be specialized to a particular type-indexed operation prior to use.

However, as usual, type classes provide access to all type-indexed operations, regardless of whether they are implemented with representation types.

For example, consider the \( \text{inc} \) operation below. It is not really important what it does, only that it depends on \( \text{zero} \) and \( \text{polymorphic equality} \). Therefore, this dependence appears in the context of \( \text{inc}R1 \) and is satisfied by making \( \text{Eq} \) and \( \text{Zero} \) superclasses of \( \text{Inc} \).

```haskell
incR1 :: (Eq a, Zero a) \Rightarrow R1 IncD a \rightarrow a \rightarrow a
incR1 a = if a \equiv zero
  then a
  else case a of
    Int1 \rightarrow a + 1
    Data1 cons \rightarrow
case findCon cons a of
  Val r kids rec \rightarrow
to emb (map l incD rec kids)
    \rightarrow a
```

```haskell
class (Eq a, Zero a, Rep1 IncD a) \Rightarrow Inc a where
  inc :: a \rightarrow a
  inc = incR1 rep1
```

Mutually recursive operations may also follow this pattern, requiring that they be superclasses of each other. However, a better pattern is to store such mutually recursive operations in the same type class. In that case, recursive dictionaries are not required and it is clear that a particular type must support both operations.

4.3 Abstract types

Suppose some type \( T \) is imported abstractly from another module. Even though we may know nothing about this type, we may still construct a representation for it.

```haskell
instance Rep T where
  rep = Data1 (DT "T" MNil) []
```
This representation includes the name of the type and the representations of any type parameters (none in this case) but otherwise contains no other information about the type. Because the structure of the type is not known, this representation cannot be used to derive instances of structurally-defined operations such as `gsum`

However, this representation is still important. First, it provides the necessary superclass context so that, if the module also exported a specialized `gsumT` operation, that operation can be used in an instance of the `GSum` type class for the type `T`.

```markdown
instance GSum T where gsum = gsumT
```

Furthermore, this representation contains just enough information for a few representation-based operations, such as `cast`, `gcast`, and the instance of `Show` for representation types.

Also, types may be represented partially. Sometimes a module may export some data constructors, but hide others. In that case, the representation can only contain the data constructors that are available.

### 4.4 Design trade-offs

There are a number of choices that occur in the design of the datatype `MTup`. Let us briefly examine the consequences of a few variations on the `R1` type (assuming that the `R` type continues to use the old definition of `MTup`).

- **Omit `Rep` a from the context**
  
  ```markdown
data MTup c l where
  MNil :: MTup c Nil
  (+::) :: c a -> MTup c l -> MTup (a ::: l)
```

  The context `Rep a` ensures that we can always convert a parameterized representation `R1 c a` to a simple representation `R a`. This means that all operations defined for type `R` are available for type `R1`. Furthermore, this context allows us to call unspecializable operations (such as `cast`) on the kids of a data constructor.

  - Include parameterized representations for all kids
    ```markdown
data MTup c l where
    MNil :: MTup c Nil
    (+::) :: c a -> MTup c l -> MTup (a ::: l)
```

  This definition would allow a type-indexed operation to ignore specializations for certain kids. It is not clear how that expressiveness would be useful. Furthermore, such representations are much more difficult to construct.

  - Include parameterized representations for some kids
    ```markdown
data MTup c l where
    MNil :: MTup c Nil
    (+::) :: Rep1 c a => c a -> MTup c l -> MTup (a ::: l)
    (-::) :: Rep a => R1 c a -> MTup c l -> MTup (a ::: l)
```

  One deficiency in the representation described in this section is that it does not extend smoothly nested datatypes. In that case, the undecidable instance declarations really are undecidable, as the type checker must satisfy ever larger type contexts.

  For example, consider the following nested datatype for perfectly balanced trees:

  ```markdown
data Sq a = L a | Br (Sq (a, a))
```

  Following the pattern described above, we define a function to construct its parameterized representation.

  ```markdown
  rSq1 :: ∀ a c. Rep a => c a -> c (Sq (a, a)) -> R1 c (Sq a)
  rSq1 c d = Data1 (DT "Sq" ((rep :: R a) ::: MNil))
             [ Con rLEmb (c ::: MNil),
              Con rREmb (d ::: MNil)]
```

  However, trouble arises if we try to use this function in the instance of the `Rep1` class. This instance requires a constraint `Sat (c (Sq (a, a)))` that can never be satisfied. (Note that it is the `Sat` constraint that causes the problem—we can create an instance of `Rep` for `Sq` in the usual manner.)

  ```markdown
  instance (Rep a, Sat (c a), Sat (c (Sq (a, a)))) => Rep1 c (Sq a) where
  rep1 = rSq1 dict
data MTup c a where
  MNil :: MTup c Nil
  (+::) :: (Rep a, Sat (c a)) => MTup c l -> MTup (a ::: l)
```

  Using the revised definition of `MTup` above, we can eliminate this unsatisfiable constraint. We do not lose any expressiveness because if a type-indexed operation uses the structure-based definition for `Sq`, it should do so for every recursive call.

```markdown
rSq1 :: ∀ a c.Rep a => c a -> R1 c (Sq a)
rSq1 d1 = Data1 (DT "Sq" ((rep :: R a) ::: MNil))
          [ Con rLEmb (d1 ::: MNil),
            Con rREmb (rSq1 d1 ::: MNil)]
instance (Rep a, Sat (c a)) => Rep1 c (Sq a) where
  rep1 = rSq1 dict
data MTup c a where
  MNil :: MTup c Nil
  (+::) :: (Rep a, Sat (c a)) => MTup c l -> MTup (a ::: l)
```

  However, although this definition allows us to create an instance of `Rep1` for `Sq`, it complicates the definitions of all type-indexed functions. Furthermore, the lack of a `Rep1` instance for `Sq` is not that limiting. Using our existing definitions, for each particular type indexed function we can still generate structure-based definitions for nested datatypes.

```markdown
instance (Rep a, GSum a, GSum (Sq a)) => GSum (Sq a) where
  gsum = gsumR1 (rSq1 dict)
```

  - Store the special cases in the context.

```markdown
data MTup c a where
  MNil :: MTup c Nil
  (+::) :: (Rep a, Sat (c a)) => MTup c l -> MTup (a ::: l)
```

  Defining type-indexed operations with the simple representations is made somewhat simpler by the fact that the representations of the kids are in the context. (For example, the definition of `nokT` in the previous section would require more manipulation of representations.)

  However, in this case, little is gained, as dictionaries must still be explicitly manipulated. Furthermore, this change comes with a loss in expressiveness. The context `Sat (c a)` says that there can be only one dictionary for the type `a`. In the next section, we discuss how the ability to have multiple dictionaries leads to greater expressiveness.

### 5. Dynamic extensibility

The previous section covered "static specialization"—a special case was incorporated into a type-indexed function at compile time. A related issue is dynamic specialization—the ability to specialize the behavior of a type-indexed function for a particular type during a particular execution of a type-indexed function.

A motivating application of dynamic specializability is type constructor analysis [9, 35, 39]. Some operations are indexed by type constructors instead of types. The key to implementing these
operations is that the type-indexed operation must temporarily treat the
argument of the type constructor in a special way.

For example, consider a generalization of “fold left” that folds
over any parameterized data structure as if it were a list.

class FL t where
foldLeft :: Rep a ⇒ (b → a → b) → (b → t a → b)
The first argument of foldLeft is actually a special case for the
type variable a of the type-indexed function lreduce below.

data LreduceD b c = LR (lreduceD :: b → c → b)
instance Lreduce b c ⇒ Sat (LreduceD b c) where
dict = LR (lreduce)
class Rep1 (LreduceD b c) ⇒ LreduceD b c where
lreduce :: b → c → b
lreduce = lreduceR1 rep1
lreduceR1 :: R1 (LreduceD b c) c ⇒ Lreduce b c
lreduceR1 (Data1 rdt cons) b c =
case (findCon cons c) of
Val rec args →
foldl1 lreduceD b rec args
lreduceR1 _ _ _ = b

The lreduceR1 function takes an argument b and returns it.
passing it through the data structure c. Importantly, a special case of
lreduce might do something different than ignore c. This is how we
define foldLeft. We embed its first argument inside a parameterized
representation and call lreduceR1 directly.

For example, the instance for trees is below. Recall that rTree1 takes two arguments. The first is the special case for the parameter
a, the second is the dictionary for Tree a. To construct the dictionary
for Tree a, we must call foldLeft recursively.

instance FL Tree where
foldLeft op =
lreduceR1 (rTree1 (LR op) (LR (foldLeft op)))

Just as foldl is used for lists, the foldLeft function can be used to
derive a number of useful operations for trees. Below are only a few examples:

gconcat :: (Rep a, FL t) ⇒ t [a] → [a]
gconcat = foldLeft (+) []
gall : (Rep a, FL t) ⇒ (a → Bool) → t a → Bool
gall p = foldLeft (λ a → b ∧ p a) True
gand :: (FL t) ⇒ t Bool → Bool
gand = foldLeft (λ) True

Note that none of these above examples are specialized to the type
constructor Tree. Any instance of the class FL may be used, and
deriving these instances only requires the analogue to rTree1.

However, there is one caveat. Spurious type class assumptions
show up in the contexts in some of these functions. For example,
gconcat requires Rep a even though this type representation is
never used. The reason for this constraint is that, for full flexibility,
the R1 GADT stores the representations of all “kid” types. This
ensures that the R1 type can always be used as an R—allowing
operations such as casting and showing the type representation.
As discussed in the previous section, an alternative is to create an
additional stripped down version of the R1 type that does not
include these representations. For simplicity we have not done so—
we need more experience to determine whether this extra constraint
is limiting in practice.

5.1 Arity 2 parameterization

Unfortunately the GADT R1 can only define type-constructor opera-
tions of arity one. Hinze [9] has noted that generalizing these
operations to multi-arithies is necessary to define operations like
fmap (requiring arity two) and zip (requiring arity three). To sup-
port such definitions in this framework requires another representa-
tion of datatypes.

data R2 c a b where
Int2 :: R2 c Int Int
Char2 :: R2 c Char Char
Arrow2 :: c a1 b1 → c a2 b2 →
R2 c (a1 → a2) (b1 → b2)
Data2 :: String → (Con2 c a b) → R2 c a b
data Con2 c a b =
∀ i1 i2.Con2 (Emb1 i1 a) (Emb1 i2 b)
(MTuple2 c i1 i2)
data MTuple2 c i1 i2 where
MNil2 :: MTuple2 c Nil Nil
(∗∗) :: c a b → MTuple2 c i1 i2
→ MTuple2 c (a ∗∗ i1) (b ∗∗ i2)

infixr 7 ∗∗;

Note that this version has been simplified, as it does not include
any Rep a constraints. Before these representations ensured that
there was enough information in this datatypes to enable operations
such as cast. However, this functionality came at the expense of
requiring Rep a for the arguments of data constructors. Instead,
the R2 representation is only intended to be used for defining
operations such as fmap, so we do not include it here.

With this infrastructure we, may define a generic map as below.

As usual, generic map is undefined for function types. (To extend
generic map to function types, we must define it simultaneously
port such definitions in this framework requires another represen-
tation of the type constructor.

instance FunRec Tree where
fmap f = mapR2 (rTree2 f (fmap f))
providing a mechanism similar to is to simplify the implementation of type-directed operations, by simply pairing a value with the representation of its type.

6.1 Dynamic typing

Dynamic typing allows type information to be truly hidden at compile time and is essential for services such as dynamic loading. It is impossible to pair a value with its representation because we cannot create a single representation that works for all type-indexed functions. Instead, true dynamic typing requires specialization mechanisms that have a dynamic semantics. For example, Washburn and Weirich demonstrate how dynamic aspects can do so in AspectML [33]. In Haskell, it is not clear how this may be done.

6.2 Pre-defined type-indexed operations

RepLib is not just a framework for defining type-indexed operations, but also a library of such operations. Some users of RepLib may never use, or even understand, representation types. Instead, they will rely on the predefined operations.

The operations in RepLib can be divided into three categories. Figure 4 lists representatives from each category. The first sort are defined for all representable types. Using these function requires merely instantiating the type class, which can be done automatically. These operations include `cast`, `eqcast` and `deepSeq` from before, as well as `subtrees`, a function that returns all kids that are the same type as its argument. The second group of operations generate instances for classes in the Haskell Prelude. These operations are already supported by deriving, but, as mentioned before, they can be used when deriving is unavailable.

Finally, RepLib includes classes with default methods. Each of these classes may be instantiated by empty instance declarations or by special cases that override the default. The functions `gsum` (from Section 4) is one of these functions, as is a specializeable version of `zero`. Other operations include a function that generates all members of a type up to a certain size, and `shrink`, a function that produces smaller versions of its argument.

The operations defined in Figure 4 are only the beginning. We hope to extend this library substantially, as well as incorporate contributions from the users of RepLib.

6.3 Expressiveness of representation types

The types `R` and `RT` defined in Sections 3 and 4 can represent many, but not all, of GHC’s types. Figure 5 summarizes. Overall, we expect that most types used by Haskell programmers will be representable, although we have not done a systematic survey. Furthermore, all types currently supported by Haskell’s `deriving` mechanism are representable.

To some extent the line in Figure 5 is not firmly drawn. It is possible to develop a more complicated type representation that would include more of the types below the line, but these modifications would entail more complexity in the definition of type-indexed operations. For example, we could enable some (but not all) higher-kinded type parameters by adding more constructors to the `MTup` datatype. We could enable some (but not all) datatypes with existential components by adding a new data constructor to the `R` type that generally represents existential binding.

In general, we have not been willing to complicate the implementation of type-directed functions so that the instances for a few esoteric types may be automatically derived. Even if a type is not representable, specific instances for it may still be explicitly provided. So, where should we draw the line? How rare are some of the types listed in Figure 5? Only practical experience can answer these questions. However, we are confident that the current definitions are a good point in the design space.

6.4 Language extensions

Although the purpose of RepLib is to eliminate boilerplate, there is still some boilerplate required in the definition of an extensible operation. As future work, we plan to consider language extensions that could simplify the definition of specializeable operations.

In particular, abstraction over type classes (similar to the proposal by Hughes [17] that was used by Lämmel and Peyton Jones [23]) could help eliminate the boilerplate of reifying type classes as explicit dictionaries. For example, in the definition of

---

### Figure 4. Some type-indexed functions of RepLib

<table>
<thead>
<tr>
<th>Representable forms</th>
<th>Unrepresentable forms</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base types</td>
<td>GADTs with existentials</td>
</tr>
<tr>
<td>Int</td>
<td>data T a where C :: b -&gt; T Int</td>
</tr>
<tr>
<td>Parameterized base types</td>
<td>data T = ∀ a.MkT (a -&gt; T)</td>
</tr>
<tr>
<td>τ1 -&gt; τ2, IO τ</td>
<td>Universal polymorphism</td>
</tr>
<tr>
<td>Newtypes</td>
<td>data T = MkT (τ a -&gt; a)</td>
</tr>
<tr>
<td>newtype T = MkT Int</td>
<td>Higher-kinded parameters</td>
</tr>
<tr>
<td>Uniform datatypes</td>
<td>data T = MkT (τ a -&gt; T)</td>
</tr>
<tr>
<td>data Nat = Z</td>
<td>Data types</td>
</tr>
<tr>
<td>Base-kind parameters</td>
<td>data T = MkT (τ a -&gt; a)</td>
</tr>
<tr>
<td>Maybe, []</td>
<td></td>
</tr>
<tr>
<td>Abstract types, void types</td>
<td>Abstract types, void types</td>
</tr>
<tr>
<td>data T</td>
<td>Abstract types, void types</td>
</tr>
<tr>
<td>Nested datatypes</td>
<td>data T a where I :: T Int</td>
</tr>
<tr>
<td>data Sq a = L a B</td>
<td>Nested datatypes</td>
</tr>
<tr>
<td>Simple GADTs</td>
<td>data Sq a = L a B</td>
</tr>
<tr>
<td>data T a where C :: b -&gt; T Int</td>
<td></td>
</tr>
<tr>
<td>Unrepresentable forms</td>
<td>data T = ∀ a.MkT (a -&gt; T)</td>
</tr>
<tr>
<td>GADTs with existentials</td>
<td>data T = ∀ a.MkT (a -&gt; T)</td>
</tr>
<tr>
<td>Existential polymorphism</td>
<td>data T = ∀ a.MkT (a -&gt; T)</td>
</tr>
<tr>
<td>Universal polymorphism</td>
<td>data T = ∀ a.MkT (a -&gt; T)</td>
</tr>
<tr>
<td>Higher-kinded parameters</td>
<td>data T = ∀ a.MkT (a -&gt; T)</td>
</tr>
<tr>
<td>data T = ∀ a.MkT (a -&gt; T)</td>
<td></td>
</tr>
</tbody>
</table>

---

### Figure 5. Expressiveness of Representation types

6. Discussion

6.1 Dynamic typing

The main application of the technology presented in this paper is to simplify the implementation of type-directed operations, by providing a mechanism similar to deriving.

However, representation types have also often been used to implement Dynamic typing [1]. Type Dynamic may be implemented by simply pairing a value with the representation of its type.

```
data Dynamic = ∀ a. Rep a ⇒ Dyn a
```

Dynamic typing allows type information to be truly hidden at compile time and is essential for services such as dynamic loading and linking. RepLib supports the operations required for dynamic typing, such as `cast` and the run-time discovery of the hidden type information through pattern matching.

However, with respect to this paper, the utility of dynamic types is limited as they cannot index specializeable operations. Even though the mechanism in Section 4 is based on representation types, resolution of special cases occurs at compile time. It is impossible to pair a value with its representation because we cannot create a single representation that works for all type-indexed functions.
gsun, we defined the type constructor GSumD to stand-in for the
type class GSum. This allowed the representation type to be
parameterized by a type class. If we had that facility natively, we
could redefine MTup as follows:

```haskell
data MTup c l where
  MNil :: MTup c Nil
  (+::) :: (Rep a, c a) ⇒ R a → MTup c l → MTup c (a ∶+: l)
```

With this version, we may define gsum, as below, with no boiler-
plate. The only difference from the non-extensible version (in Sec-
tion 3) is the use of the R1 type and the recursive call through the
type class.

class Rep1 GSum a ⇒ GSum a where
  gsum :: a → Int
  gsum = gsumR1 rep1

```haskell
  gsumR1 :: R1 GSum a → a → Int
  gsumR1 Int x = x
  gsumR1 (Arrow1 _, _) x = error "you"

  gsumR1 (Data1 dt cons) x =
  case (findCon cons x) of
    Val emb rec kids →
      (foldL1 λ (a b → gsum b + a) 0 rec kids
    gsumR1 _ _ _ = 0
```

However, to define operations like foldLeft in the presence of
class parameterization, we must be able to specify an alternate dic-
tionary to be included in the representation. Named type class in-
stances [19] would allow that behavior. Other language extensions
that we plan to consider are mechanisms to support dynamic spe-
cialization of type-indexed functions, as we briefly mentioned in
Section 6.1, and a uniform treatment of kind-indexed types, so that
we may do a better job with higher-kind type constructors.

### 7. Related work

Representation types were first introduced in the context of type-
preserving compilation [4]. However, because they provide a clean
way to integrate run-time type analysis into a language with a type-
erasure semantics, Cheney and Hinze [2] showed how to encode
them in Haskell 98 using a derived notion of type equivalence. Rep-
resentation types may also be implemented with a Church encoding
[34]. However, in our view GADTs provide the best program-
mapping model for representation types: they support simple defini-
tions of type-indexed functions via pattern matching and GADT
type refinement automatically propagates the information gained
through this matching without the use of type coercions.

The idea (in Section 2) of using a type class to automatically
provide type representations also appears in Cheney and Hinze’s
First-class phantom types [2]. However, that paper does not use a
default class method, enabling the class to limit the domain of
the type-indexed operation. Instead they create a generic instance that
provides the type-indexed operation for all representable types.

The Rep class is similar to GHC’s Typeable class, except that
Rep uses a GADT for the type representation and Typeable uses
a normal datatype. Functions defined with Typeable therefore re-
quire more uses of cast as there is no connection between argu-
ments and their type representations. Furthermore, in GHC, the
Typeable class may only represent uniform (non-nested) datatypes,
that do not contain existential components, that are not GADTs,
and that are only parameterized by constructors of base kind. In
contrast, the Rep class includes all the above as well as nested
datatypes and some GADTs.

The Typeable class type is the foundation for the “Scrap your
boilerplate” library [21, 22, 23]. This library includes a num-
ber of combinators for assembling type-indexed functions from
smaller components. This style of programming is compatible with
RepLib—in fact we were able to port a module of traversal schemes
(such as everywhere) to RepLib by renaming a single type class.

The idea of generically representing data constructors via iso-
morphisms (in Section 3) was first used by Generic Haskell and
Derivable Type Classes [15], where data constructors were com-
piled to binary sums and products. It first saw specific use with rep-
resentation types in an unpublished manuscript [37, 39] that made
data constructors isomorphic to n-tuples. Recently Hinze, Löh and
others [13, 12, 16] have devised many more generic views of data
types, and provide a detailed comparison of these views. How-
ever, the specific isomorphism between data constructors and list of
types is new to this paper. All of these isomorphisms provide sim-
ilar expressive power—however, we think that manipulating type
lists, either natively or with folds and maps, provides the most nat-
ural definition of type-indexed operations.

Derivable type classes [15] is closely related to the work de-
scribed here. Like Generic Haskell, this approach treats datatypes
as isomorphic to sums of products. However, as Lämmel and Pey-
ton Jones [23] point out, programming with datatypes in this man-
ner is tricky to get right. Furthermore, derivable type classes require
much more specific help from the compiler—the implementation of
a domain specific language for specifying how derivable instances
should be generated.

The idea of parameterizing a representation type to allow type-
constructor analysis (Section 5) first appeared in the authors PhD
thesis [36], and application to Haskell representation types first
appeared in the manuscript mentioned above [37]. In Generics for
the Masses (GM) [10], Hinze translated this code to use type
classes instead of first-class polymorphism, enabling it to be used
with Haskell 98.

The idea that this same parameterization could be used to en-
able extensible type-indexed operations (Section 4) is new to this
paper. It was inspired by the third “Scrap Your Boilerplate” pa-
er of Lämmel and Peyton Jones [23], although the mechanism in
that paper is quite different. One difference is that SYB3 relies on
overlapping instances that automatically enable type-indexed func-
tions for all types. Although overlapping instances are convenient,
they do not permit the designers of type-indexed functions to limit
their domains to a particular set of types. Furthermore, overlapping
instances require careful thought about the context reduction algo-
rumth to ensure that appropriate instances are chosen in each case.
For these reasons, we have not used overlapping instances.

The ideas of Section 4 have been concurrently explored in the
context of the GM framework [27]. Furthermore in the extended ver-
sion of Scrap your Boilerplate Reloaded, Hinze and Löh [14] de-
scribe an extensible version of spine-based generic programming.
Both of these provide a different programming model for type-
indexed functions.

### 8. Conclusion

More than these individual ideas, the contribution of this paper is
the RepLib library that combines them together in a coherent for-
mat. We intend to distribute and maintain this library, and accu-
mulate new examples of type-indexed operations. Although this li-
brary is specific to GHC, we hope that the extensions that it relies
on—GADTs, scoped type variables, higher-rank polymorphism,
and more flexible instance declarations—will be adopted by future
Haskell compilers.

### Acknowledgments

Thanks to the anonymous reviewers of the Haskell Workshop com-
mittee. This document was produced from literate Haskell source
References


