RepLib: A library for derivable type classes

Stephanie Weirich
University of Pennsylvania

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.

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 Eq type class defines the signature of polymorphic equality.

```
class Eq a where (≡) :: a → a → Bool
```

The instances of the Eq class define the behavior of polymorphic equality at specific types. For example, an instance for the data type Tree is below.

```
data Tree a = Leaf a | Branch (Tree a) (Tree a)

instance Eq a ⇒ Eq (Tree a) where
  (Leaf x1) ≡ (Leaf x2) = x1 ≡ x2
  (Branch t1 t2) ≡ (Branch s1 s2) = t1 ≡ s1 ∧ t2 ≡ s2
  _ ≡ _ = False
```

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 \( Eq \).

However, Haskell programs often include many datatype definitions and it can be tiresome to define instances of \( Eq \) for all of these types. Furthermore, there is often a relationship between the structure of a datatype definition and its instance for \( Eq \), so many of these instances have similar definitions.

As a result, the Haskell language includes the deriving mechanism that can be used to direct a Haskell compiler to insert an instance of the \( Eq \) based on the structure of a newly defined datatype. For example, the code above may be replaced by the following.

```
data Tree a = Leaf a | Branch (Tree a) (Tree a)
deriving (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 \( Eq \) when directed, but allowing programmers to write their own specialized instances when necessary.

Unfortunately, deriving only works for a handful of built-in type classes. In Haskell 98, only \( Eq, Ord, Bounded, Show \) and \( Read \) 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 Polypoly Programming (PolyP) [18], Generic Haskell [3, 24], Derivable type classes [11], the 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]. This library 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 arbitrary 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 one is essential to the design of the library. 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).
- RepLib allows type-indexed operations to be defined by representation types, but called via type classes. This means that type classes can automatically supply the representation type arguments and restrict the domain of type-indexed functions.
- To generically represent the structure of datatypes with a list of data constructor embeddings (Section 3).
- In RepLib, newtypes and datatypes are represented structurally. This, in combination with default class methods, provides a way to automatically derive the behavior of a type-indexed operation for a new datatype with only an empty instance declaration.
- To parameterize the representation of datatypes with explicit dictionaries (Section 4).

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Copyright © ACM 0-89791-88-6/97/05 . $5.00.
The RepLib library also includes extensible representations that allow programmers. The default behavior of a type-indexed operation defined by an extensible representation may be overridden at a specific type.

- To support the definition of functions indexed by parameterized types by allowing dictionaries to be dynamically supplied (Section 5).

Type-indexed operations, such as reductions, require this mechanism.

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 compatible 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-scoped 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 and many have been around for years.

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 the example is to introduce representation types and clarify the roles that they and type classes should play. The code developed in this section 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 can define the representation type \( R \) below, following GADT notation, by listing all of its data constructors with their types.

```haskell
data R a where
  Int :: R Int
  Unit :: R ()
  Bool :: R Bool
  Char :: R Char
  Pair :: R a -> R b -> R (a,b)
  Arrow :: R a -> R b -> R (a -> b)
  List :: R a -> R [a]
```

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 \( R \text{Int} \), then we know that that 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").

\[
\begin{align*}
gsumR :: & R a -> a -> R Int \\
gsumR \text{Int} x = & x \\
gsumR \text{(Pair t1 t2)} (x1,x2) = & gsumR t1 x1 + gsumR t2 x2 \\
gsumR \text{(List t)} l = & \text{foldl} \, (\lambda s x \rightarrow (\text{gsumR t} \, x) + s) \, l \\
gsumR \_ x = & 0 \\
\end{align*}
\]

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 destroy the argument. Likewise, in the \( \text{List} \) branch, \( l \) must be a list and so is an appropriate argument for \( \text{foldl} \).

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,

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

At this point, it is useful to compare the definition of \( gsumR \) with a type-class based implementation. We could rewrite the generic sum function using type classes as:

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

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

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

Defining generic sum with type classes loses the simple notation of pattern matching 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 anytime 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

\(^1\)http://www.cis.upenn.edu/~swierich/RepLib
type classes make it easier to use. However, by using type classes and representation types together, we can get the advantages of both.

Consider a class \texttt{Rep} that includes all types that are representable.

\begin{verbatim}
class Rep a where rep :: R a
\end{verbatim}

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

\begin{verbatim}
instance Rep Int where rep = Int
instance Rep () where rep = Unit
instance Rep Bool where rep = Bool
instance Rep Char where rep = Char
instance (Rep a, Rep b) ⇒ Rep (a, b) where rep = Pair rep rep
instance (Rep a, Rep b) ⇒ Rep (a → b) where rep = Arrow rep rep
instance (Rep a) ⇒ Rep [a] where rep = List rep
\end{verbatim}

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

\begin{verbatim}
class Rep a ⇒ GSum a where
gsum :: a → Int
gsum = gsumR rep
\end{verbatim}

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 \texttt{gsumR}. Instead, the instances “derive” the definition of \texttt{gsum} for these particular types.

\begin{verbatim}
instance GSum Int instance GSum () instance GSum Bool instance GSum Char instance (GSum a, GSum b) ⇒ GSum (a, b) instance GSum a ⇒ GSum [a]
\end{verbatim}

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 \texttt{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 \texttt{gsum} correctly.

However, we have so far gained very little. The extensibility problem still 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 \texttt{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 \texttt{gsumR} based on the structure of the represented type. In particular, we would like to define \texttt{gsumR} for both \texttt{Pairs} and \texttt{Lists} with the same code instead of writing two similar branches.

In this section, we describe a representation type that can generically represent the structure of all Haskell 98 datatypes. For reference, the infrastructure developed below and in Section 4 appears in the Appendix. Consider the following revised definition of the \texttt{R} type:

\begin{verbatim}
data R a where
  Int :: R Int
  Char :: R Char
  Arrow :: R a → R b → R (a → b)
  Data :: DT → [Con a] → R a
\end{verbatim}

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

The \texttt{Data} constructor takes two arguments: information about the data type itself \texttt{DT} and information about each of the data constructors that make up the datatype (the list of \texttt{Con} \texttt{a}). There are many possibilities for the generic representation of datatypes and deciding what should be included involves a number of trade-offs. In Section 3.1 below, we begin our discussion with the design of \texttt{Con} and then in Section 3.2 we cover \texttt{DT}. We also return to this topic in Section 7, where we compare this design to previous work.

#### 3.1 Representing data constructors

We start our discussion with the definition of the type \texttt{Con}, the representation of a data constructor. What information should be included about data constructors such as \texttt{Leaf} or \texttt{Branch}?

First, the representation should include the types of the arguments to the data constructor (also called the “kids” of the datatype.) For example, the representation of \texttt{Branch} should record that \texttt{Branch} has two kids, both of type \texttt{Tree a}. Furthermore, because data constructors each take different types of arguments, some sort of existential quantification must hide these types. Moreover, because data constructors each take different numbers of arguments, this existential must hide varying numbers of types, aggregated together in some structure. Therefore, we use type lists (also called heterogeneous lists [20]) as that structure, defined by the following two single-constructor datatypes. (By convention, the type variable \texttt{a} stands for an arbitrary type, while the type variable \texttt{l} stands for a type list. However, there is nothing in this definition that enforces that usage.)

\begin{verbatim}
data Nil = Nil
data a * l = a * l
infixr 7 *
\end{verbatim}

Type lists generalize n-tuples. For example, the type \((\text{Int} * \text{Char} * \text{Nil})\) is isomorphic to the pair type \((\text{Int}, \text{Char})\).

\begin{verbatim}
example1 :: (Int * Char * Nil)
example1 = 2 * ‘b’ * Nil
\end{verbatim}

As well as defining type lists, we must also represent them. We do so with the following GADT. Like the \texttt{R} type, the type index describes what type (list) the term represents. The \((\oplus)\) constructor includes \texttt{Rep} \texttt{a} in its context so that, as this list is destructed, this representation may be implicitly provided. Sometimes it is impossible to unambiguously provide this implicit representation, so the constructor also includes an explicit representation \texttt{R a} for those situations.

\begin{verbatim}
data Rs l where
  RNil :: Rs Nil
  (⊕) :: Rep a ⇒ Rs l → Rs (a * l)
infixr 7 ⊕
\end{verbatim}

\begin{verbatim}
examp1 :: Rs (Int * Char * Nil)
examp1 = Int ⊕ Char ⊕ RNil
\end{verbatim}

In this way, the type \texttt{Rs l} represents a list of types, but we do not know exactly what those types are or even how many of them there
are. As with the $R$ type, we define a type class to automatically provide the representation of a list of types.

```haskell
class Reps l where
    reps :: Rs l
instance Reps Nil where
    reps = RNil
instance (Rep a, Reps l) ⇒ Reps (a * l) where
    reps = rep ⊗ reps
```

The second ingredient we need in the representation of a data constructor that produces an $a$ 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 data constructor, and extract its arguments. Also, given arguments of the appropriate types, we should be able to construct an $a$.

Therefore, we represent a data constructor as the representation of its lists of arguments and a embedding-projection pair between the arguments of the constructor and the datatype. The $Emb$ record below contains a generic version of a constructor and a generic deconstructor.

```haskell
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:

```haskell
rLeafEmb :: Emb (a * Nil) (Tree a)
{ to = λ(a * Nil) → (Leaf a),
  from = λx → case x of
  Leaf a → Just (a * Nil)
  _ → Nothing }
```

Finally, we may use $Emb$ to define the representation of an arbitrary data constructor of the datatype $a$ as an embedding between some representable type list $l$ and $a$.

```haskell
data Con a = ∀ l. Reps l ⇒ Con (Emb l a)
```

The $∀$ in the definition of $Con$ means that its type 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, this type is existentially quantified. Furthermore, the class constraint carries the representation of the type list. To form the representations of the data constructors, we need the representation of the type $a$ to satisfy the class constraint of $Con$.

```haskell
rLeaf :: Rep a ⇒ Con (Tree a)
rLeaf = Con rLeafEmb
rBranch :: Rep a ⇒ Con (Tree a)
rBranch = Con rBranchEmb
```

The above definition of $Con$ contains only the minimum information required for representing data constructors. For operations such as reading and showing, this representation should also include additional information 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. At a minimum, this should include the name of the datatype and the representations of its parameters.

```haskell
data DT = ∀ l. DT String (Rs l)
```

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

```haskell
instance Rep a ⇒ Rep (Tree a) where
    rep = Data (DT "Tree" ((rep :: R a) ⊗ RNil)) [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. For example, displaying the representation of types such as $Tree$ $Int$ or $Tree$ $Bool$ requires the name “Tree” and the representation of the type parameters $Int$ and $Bool$.

For example, the instance of $Show$ below displays a representation type. Note that pattern matching allows a natural definition for showing the list of type parameters.

```haskell
instance Show (R a) where
    show Int = "Int"
    show Char = "Char"
    show (Arrow r1 r2) = "(" ++ (show r1) ++ " -> " ++ (show r2) ++ ")"
    show (Data (DT str reps) _) = "(" ++ str ++ " show reps ++ ")"
```

```haskell
instance Show (Rs l) where
    show RNil = ""
    show (r ⊗ RNil) = "r ++ 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 $DT$ information also means that a type-safe cast [38] of type

```haskell
cast :: (Rep a, Rep b) ⇒ a → Maybe b
```

and the related generalized cast

```haskell
gcast :: (Rep a, Rep b) ⇒ c a → Maybe (c b)
```

can be implemented. Without this information, these operations cannot enforce the distinction between isomorphic type. While the basic cast may be implemented directly, the implementation of the generalized cast requires the use of an unsafe type coercion. However, for practical reasons, basic cast is also implemented with $primUnsafeCoerce#$ in the accompanying distribution.

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. Furthermore, given the definition of such datatypes (except for GADTs), the accompanying distribution includes Template Haskell code to automatically generate its representation and instance declaration for the $Rep$ type class.

### 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:
with a type list, we can define a library of folds and maps for type
constructor. By representing the arguments to a type constructor
the results together.

The new part of this example is the case for

Data

findCon

will be able to decompose the type, and we will never get to the
rectly represented the datatype, then one of the generic destructors

Note that

findCon

error

case

of

Just kids → gsumRL reps kids
Nothing → findCon rest

The new part of this example is the case for Data. Given an
argument of type a, the auxiliary function findCon iterates through
the data constructors until it finds the appropriate one and then
calls gsumR on all of the arguments to this constructor, adding
the results together.

Note that findCon does not have a case for []. If we have cor-
rectly represented the datatype, then one of the generic destructors
will be able to decompose the type, and we will never get to the
end of this list. This looping pattern appears often in type-indexed
code, so it makes sense to factor it out. Below, we define the func-
tion findCon that performs this loop. The result of this function
must existentially bind the type list—so we also define a new data
constructor Val that contains the arguments of the data constructor,
the representation of their types, and the embedding-projection pair
for that data constructor.

data Val a = ∀ l. Reps l ⇒ Val (Emb l a) l
findCon :: [Con a] → a → Val a
findCon (Con emb : rest) x = case (from emb x) of
  Just kids → Val emb kids
  Nothing → findCon rest x

Furthermore, once we have found the appropriate data construc-
tor the next computation is often a fold over the arguments to that
constructor. By representing the arguments to a type constructor
with a type list, we can define a library of folds and maps for type
lists. For example, the analogues of foldl and map appear below.

foldl₁ :: (∀ a.Rep a ⇒ a → b → b) → b
        Rs l → l → b
foldl₁ f b RNil Nil = b
foldl₁ f b (⊕ rs) (a * l) = f a (foldl₁ f b rs l)
map₁ :: (∀ a.Rep a ⇒ a → a) → Rs l → l → l
map₁ t RNil Nil = Nil
map₁ t (⊕ rs) (a * a₁) = (t a * map₁ t rs a₁)

With these operations, we can rewrite the Data branch for
gsumR more succinctly as shown in Figure 1. (Note that because
of the existential component of Val, we must use case instead of
let to pattern match the result of findCon.) This Figure is
the complete definition of generic sum, including the type class
definition discussed in the previous section. If a programmer
would like to derive an instance of GSum for a new type, he need only
make sure that the representation of that type is available and then
create the trivial instance of GSum for the new type.

“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
each example, one part of the SYB library defines generic traversals
over datatypes, using the type-indexed operations mapT, foldT
and everywhere. Below, we show how to implement those opera-
tions with representation types.

A traversal is a function that has a specific behavior for a par-
ticular 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 monomor-
phic function of type t → t to be a Traversal.

mkT :: (Rep a, Rep b) ⇒ (a → a) → b → b

mkT f (Rep a, Rep b) ⇒ (a → a) → b → b

Next, the mapT function extends a basic traversal to a “one-layer”
traversal. It maps the traversal across the subcomponents of a data
constructor. In this setting, mapT has the following definition.

Note that the wildcard pattern match and library routines findCon
and mapT provide a very concise definition of this type-indexed
function. Also 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 str cons) → λx →
  (findCon cons x) of
    Val emb kids → to emb (map₁ t reps 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
everywhere f x = f (mapT (everywhere f) x)

With these operations we can compile and execute the “par-
adise” 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, given the inter-
esting case for increasing salaries.

---

Figure 1. Generic Sum

gsumR :: R a → a → Int

gsumR Int x = x

gsumR (Arrow r1 r2) f = error "urk"

gsumR (Data rdt cons) x =
case (findCon cons x) of
  Val emb kids → foldl₁ (λa b → (gsumR rep a) + b) 0 reps kids

-- Type class with default definition
class Rep a ⇒ GSum a where
gsum :: a → Int

gsum = gsumR rep

-- Enable gsum for common types
instance GSum Int
instance GSum Bool
  etc...

foldl₁ :: (∀ a.Rep a ⇒ a → b → b) → b
        Rs l → l → b
foldl₁ f b RNil Nil = b
foldl₁ f b (⊕ rs) (a * l) = f a (foldl₁ f b rs l)
map₁ :: (∀ a.Rep a ⇒ a → a) → Rs l → l → l
map₁ t RNil Nil = Nil
map₁ t (⊕ rs) (a * a₁) = (t a * map₁ t rs a₁)
increase :: Float -> Company -> Company
increase k = everywhere (mkT (incSk k))

incSk :: Float -> Salary -> Salary
incSk k (S s) = S (s * (1 + k))

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

However, representation types are sometimes more natural to program with than the SYB library or spines. For example, polymorphic equality requires a “twin-traversal” scheme in SYB [22]. With spines, it must be generalized to compute equality between arguments of two different types. Using representation types we can express this operation more naturally:

```
  eqR :: R a -> a -> a -> Bool
  eqR Int = (=)
  eqR Char = (=)
  eqR (Arrow t1 t2) = error "urk"
  eqR (Data1 cons) = lambda y -> loop cons x y
  where
      loop (Con1 emb : rest) x y =
      case (from emb x, from emb y) of
        (Just p1, Just p2) -> eqRL reps p1 p2
        (Nothing, Nothing) -> loop rest x y
        (x, y) -> False

  eqRL :: Rs l1 -> l1 -> l1 -> Bool
  eqRL RNil RNil RNil = True
  eqRL (r ⊕ r's) (p1 ⊕ t1) (p2 ⊕ t2) = eqR r p1 p2 ∧ eqRL rs t1 t2
```

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.

```
  instance (Rep a, Eq a) => Eq (Tree a)
  where
    (=) = eqR rep
```

We might create such an instance when deriving is not available—for example, if we did not have access to 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.

4. 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. Below, we explain this issue in more detail and describe an extension of the R type that allows type-indexed operations to be specialized for specific types.

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.

```
newtype IntSet = IS [Int]

rSEmb :: Emb ([(Int) * Nil]) IntSet
rSEmb = EmbNil

iSEmb = EmbNil = λ(Ñd * Nil) → IS d,
from = λ(IS d) → Just (Ñd * Nil)
```

--- An explicit dictionary for the type class

```
data GSumD a = GSumD{ gsumD :: a -> Int }

instance GSum a => Sat (GSumD a) where
dict = GSumD gsum
```

--- Type structure based definition

```
gsumR1 :: R1 GSumD a -> a -> Int
gsumR1 Int1 x = x
```

```gsumR1 Arrow1 r1 r2) f = error "urk"
gsumR1 (Data1 dt cons) x =
  case (findCon1 cons x) of
    Val1 emb kids rec →
      foldlI f (λca a b → (gsumD ca b) + a) 0 rec kids
```

--- Class with default definition

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

--- Enable gsum for common types

```
instance GSum Int
```

--- Special case for sets

```
instance GSum IntSet where
gsum (IS 1) = gsum (nub 1)
```

--- Special case for IntSet

```
instance Rep1 IntSet where

  rep = Data (DT "IntSet" RNil) [Con rSEmb]
```

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.

```
instance GSum IntSet where
gsum (IS 1) = gsum (nub 1)
```

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).

```
Main > gsum (IS [1,1])
1
Main > gsum (Leaf (IS [1,1]))
2
```

4.1 Parameterized representations

The key idea to extending type-indexed functions with specific functionality for particular types 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. These dictionaries are stored in theMTup c ld component of the data constructor representation, Con1, below. (Note that new definitions in this section end with 1 to distinguish them from those of the previous section.)
data Con1 c a =
    ∀ tl. Con1 (Emb tl a) (MTup c tl)

MTup is a GADT that, like Rs, is indexed by a type list. For every
type in that list, this data structure stores its representation
(implicitly) and a dictionary for a particular type class.

data MTup c l where
    MNil :: MTup c Nil
    (o) :: (Rep a) ⇒ c a → MTup c l → MTup c (a * l)

The dictionary may be for any type-indexed operation. Therefore,
we parameterize the types Con1, MTup and R1 with the type of
the dictionary, c. A representation of type R1 c a may only be used
to define a type-indexed operation of type c a. Otherwise, there is
no change to the representation type.

data R1 c a where
    Int1 :: R1 c Int
    Char1 :: R1 c Char
    Arrow1 :: (Rep a, Rep b) ⇒ c a → c b → R1 c (a → b)
    Data1 :: DT → [Con1 c a] → R1 c a

As before, we create a (now multiparameter) type class to automatica-
ly supply type representations. So that we may continue to support
all previous operations, such as cast, we make this class a subclass
of Rep.

class Rep a ⇒ Rep1 c a where
    rep1 :: R1 c a

The success of this scheme depends on the fact that representa-
tions are created with care. A function to create representation
types must abstract the contexts that should be supplied for each of
the kids. For example, the representation of Trees below abstracts
the explicit dictionaries ca and ct for the type parameters a and the
type Tree a that appear in the kids of Leaf and Branch.

rTree1 :: ∀ a c.
    (Rep a) ⇒ c a → c (Tree a) → R1 c (Tree a)

rTree1 ca ct =
    Data1 (DT "Tree" ((rep :: R a) ⊕ RNil))
        [Con1 rLeafEmb (ca ⊕ MNil),
         Con1 rBranchEmb (ct ⊕ ct ⊕ MNil)]

It is the job of the instance declaration that automatically
creates the representation of the tree type to supply these dictionari-
es. These dictionaries are provided by instances of the type class
Sat. This type class can be thought of as a “singleton” type class,
the class of types that contain a single value.  

class Sat a where
    dict :: a

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

instance (Rep a, Sat (c a), Sat (c (Tree a))) ⇒
    Rep1 c (Tree a) where
    rep1 = rTree1 dict

Likewise, the representation of IntSet requires an instance of
Sat for its kid, [Int].

\footnote{In fact, the type R t is also a singleton type for any t. Therefore, we could replace the class Rep with Sat (R a). For simplicity, we do not do so.}

instance Sat (c [Int]) ⇒ Rep1 c IntSet where
    rep1 = Data1 (DT "IntSet" RNil)
        [Con1 rSEmb (dict ⊕ 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 specilizability. Consider the definition of a spe-
cializable version of generic sum, shown in Figure 2. (This func-
tion relies on findCon1, Val1, and foldl₁l that are the analogues
to the auxiliary functions defined in the previous section.) The first
step is to create a dictionary for this operation and a generic
instance declaration for 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 gsumD that
accesses the stored dictionary instead of calling gsumR1 directly.
In fact, we cannot call gsumR1 recursively, as Con1 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 IntSets.

Main > gsum (IS [1, 1])
1
Main > gsum (Leaf (IS [1, 1]))
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 in-
crease is eligible. Some frameworks for type-indexed programming
do have difficulty with this situation. One might think that is also
the case here, as the parameterized representation type must be spe-
cialized 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, suppose there exists a class
Zero that provides an arbitrary element zero for any type.

Now, consider the inc operation below. It is not really important
what it does, only that it depends on zero and polymorphic equality.
Therefore, this dependence appears in the context of incR and is
satisfied by Eq and Zero superclasses of Inc.

incR1 :: (Eq a, Zero a) ⇒ R1 IncD a → a

then a
else case r of
    Int1 → a + 1
    Dat1 _ cons →
    case findCon1 cons a of
        Val1 emb kids rec →
            to emb (map IncD rec kids)
        _ → a

class (Eq a, Zero a, Rep1 IncD a) ⇒ Inc a where
    inc :: a → a
    inc = incR1 rep1

Mutually recursive operations also follow this pattern, requiring
that they be superclasses of each other.

class (Rep1 FooD a, Bar a) ⇒ Foo a where
    foo :: ...
    foo = fooR rep1
There are a number of choices that occur in the design of the representation can only contain the data constructors that are may export some data constructors, but hide others. In that case, the instance of \( \text{Show} \) for a few representation-based operations, such as \( gsum \), the necessary superclass context so that, if the module also exported a specialized \( gsum T \) operation, that operation can be used in an instance of the \( \text{GSum} \) type class for the type \( T \).

### 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 = Data (DT "\( T \)" RNil) []
```

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 \( gsum T \) operation, that operation can be used in an instance of the \( \text{GSum} \) type class for the type \( T \).

```haskell
instance GSum T where gsum = gsum T
```

Furthermore, this representation contains just enough information for a few representation-based operations, such as \( \text{cast} \), \( \text{gcast} \), and the instance of \( \text{Show} \) 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 \( \text{MTup} \) above. Let us briefly examine the consequences of a few variations.

- **Omit \( \text{Rep} a \) from the context**

  ```haskell
data MTup c l where
MNil :: MTup c Nil
(\( \odot \)) :: c a \( \rightarrow \) MTup c l \( \rightarrow \) MTup (a \( * \) l)
```

  By including the simple representations, we enable some uses that would not otherwise be supported. For example, some examples from the SYB library require type casting, which is only supported by the simple representations. By making sure that the parameterized representation is also an extension of the simple representation, we ensure that we always have the capabilities of the simpler representation.

- **Include parameterized representations for all kids**

  ```haskell
data MTup c l where
MNil :: MTup c Nil
(\( \odot \)) :: \( \text{Rep1} \) c a \( \Rightarrow \) c a \( \rightarrow \) MTup c l \( \rightarrow \) 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.

- **Include parameterized representations for some kids**

  ```haskell
data MTup c l where
MNil :: MTup c Nil
(\( \odot \)) :: \( \text{Rep} \) a \( \Rightarrow \) c a \( \rightarrow \) MTup c l \( \rightarrow \) MTup (a \( * \) l)
```

  (\( \odot \)) :: \( \text{Rep} \) a \( \Rightarrow \) R1 c a \( \rightarrow \) MTup c l \( \rightarrow \) 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 undeclarable instance declarations really are undeclarable, as the type checker must satisfy ever larger type contexts.

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

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

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

  ```haskell
rSq1 :: \( \forall \) a c. 
  Rep1 a \( \Rightarrow \) c a \( \rightarrow \) (Sq (a, a)) \( \rightarrow \) R1 c (Sq a)
  rSq1 c d = Data1 (DT "\( \text{Sq} \)" [(\( \text{rep} :: \text{R} \) a) \( \odot \) RNil]))
  [Con1 rLEmb (c \( \odot \) MNil),
   Con1 rBREmb (d \( \odot \) MNil)]
```

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

  ```haskell
instance (Rep a, Sat (c a), Sat (c (Sq (a, a)))) \( \Rightarrow \)
  Rep1 c (Sq a) where
  rep1 = rSq1 dict dict
```

  Using the revised definition of \( \text{MTup} \) 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 \( \text{Sq} \), it should do so for every recursive call.

  ```haskell
rSq1 :: \( \forall \) a c. Rep a \( \Rightarrow \) c a \( \rightarrow \) R1 c (Sq a)
  rSq1 d1 = Data1 (DT "\( \text{Sq} \)" [(\( \text{rep} :: \text{R} \) a) \( \odot \) RNil]))
  [Con1 rLEmb (d1 \( \odot \) MNil),
   Con1 rBREmb (rSq1 d1 \( \odot \) MNil)]
```

- **Include parameterized representations for some kids**

  ```haskell
instance (Rep a, Eq a, Eq (Sq a)) \( \Rightarrow \)
  Eq (Sq a) where
  eq = eqR1 (rSq1 dict dict)
```

- **Store the special cases in the context.**

  ```haskell
data MTup c a where
  MNil :: MTup c Nil
  (\( \odot \)) :: (Rep a, Sat (c a)) \( \Rightarrow \)
  MTup c l \( \rightarrow \) MTup c (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 \( \text{nkT} \) 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 for a particular type 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 the first argument of the type-indexed operation must temporarily treat the type variable \( a \) as if it were a list.

```plaintext
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 LRReduces b c = LR { lreduceD :: b → c → b }

instance Reduce b c ⇒ Sat (LRReduces b c) where
dict = LR (lreduce)

class Rep1 (LRReduces b) c ⇒ Reduce b c where
    lreduce :: b → c → b
    lreduce = lreduceRep1 rep1

lreduceRep1 :: (LRReduces b) c ⇒ Reduce b c where

lreduce1 :: (LRReduces b) c → Reduce b c

lreduceRep1 :: (Data1 rdt cons) b c where

case (finalCon1 cons c) of
    Val1 rcd args rec →
        foldl1 lreduceD b rec args
    lreduce1 _ b c = b

The lreduce1 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 lreduceRep1 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 lreduce recursively.

instance FL Tree where

foldLeft op =
    lreduceRep1 (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 x 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 “kind” 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 operations of arity one. Hinze [9] has noted that generalizing these operations to multi-ary types is necessary to define operations like fmap (requiring arity two) and zip (requiring arity three). To support such definitions in this framework requires another representation 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 =
    ∀ l1 l2.Con2 (Emb l1 a) (Emb l2 b) (MTup2 c l1 l2)

data MTup2 c l1 l2 where

    MI2 rcd2 Nil Nil
    (* *) :: c a b → MTup2 c l1 l2
    → MTup2 c (a * l1) (b * l2)

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 with its inverse [25].) For datatypes, generic map iterates the mapping function over the kids of the data constructor. For all other base types, generic map is an identity function.

mapR2 :: R2 (→) a b → a → b
mapR2 (Arrow2 _ _) = error "urk"
mapR2 (Data2 rdt cons) = λ x →
    let loop (Con2 rcd1 rcd2 ps : rest) =
        case from rcd1 x of
            Just a → to rcd2 (mapR2 ps a)
            Nothing → loop rest
    in loop cons
mapR2 Int2 = id
mapR2 Char2 = id
mapR2 :: MTup2 (→) l1 l2 → l1 → l2
### Representable forms

<table>
<thead>
<tr>
<th>Base types</th>
<th>Int</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameterized base types</td>
<td>$\tau_1 \to \tau_2$, IO $\tau$</td>
</tr>
<tr>
<td>Newtypes</td>
<td>newtype $T = MkT$ Int</td>
</tr>
<tr>
<td>Uniform datatypes</td>
<td>data Nat = Z</td>
</tr>
<tr>
<td>Base-kind parameters</td>
<td>Maybe, []</td>
</tr>
<tr>
<td>Abstract types, void types</td>
<td>data $T$</td>
</tr>
<tr>
<td>Nested datatypes</td>
<td>data $Sq a = L a \mid B (Sq (a, a))$</td>
</tr>
<tr>
<td>Simple GADTs</td>
<td>data $a$ where $I :: T$ Int</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Unrepresentable forms</th>
</tr>
</thead>
<tbody>
<tr>
<td>GADTs with existentials</td>
</tr>
<tr>
<td>Existential polymorphism</td>
</tr>
<tr>
<td>Universal polymorphism</td>
</tr>
<tr>
<td>Higher-kind parameters</td>
</tr>
</tbody>
</table>

**Figure 3.** Expressiveness of Representation types

```haskell
mapRL2 MNil2 Nil = Nil
mapRL2 (f *** rs) (a * l) = f a * mapRL2 rs l
```

The arity-2 representation of a type constructor is similar to the arity-1 representation, and may also be automatically generated. For example, the definition of $rTree2$ is below.

```haskell
rTree2 :: \forall a b c c a b c \to c (Tree a) (Tree b) 
\to R2 c (Tree a) (Tree b)
```

The implementation of $mapR2$ and the representation of $Tree$ derives an instance for the `Functor` constructor class.

```haskell
instance Functor Tree where
  fmap f = mapR2 (rTree2 f (fmap f))
```

### 6. Discussion

Below, we discuss three remaining questions about representation-type based programming: How do the mechanisms discussed in this paper interact with dynamic typing? What types can actually be represented? What extensions to Haskell would simplify or enhance programming with representation types?

#### 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]. `Dynamic` may be implemented simply by pairing a value with the representation of its type.

```haskell
data Dynamic = \forall a. Rep a \Rightarrow 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 `specializable` 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 $R1$ representation because we cannot create a single $R1$ representation that works for all type-indexed functions.

Instead, true dynamic typing requires specialization based on 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 Expressiveness of Representation Types

The types $R$ and $R1$ defined in Sections 3 and 4 can represent many, but not all, of GHC’s types. Figure 3 summarizes what is and is not possible. Overall, we expect that most types used by Haskell programmers will be representable, although we have not done a systematic survey.

To some extent the line in Figure 3 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-kind type parameters by adding more constructors to the $R$s 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, and these instances will be used if that type appears in the structure of some other type.

So, where should we draw the line? How much complication should we allow? How rare are some of the types listed in Figure 3? Only practical experience can answer these questions. However, we are confident that the current definitions are a good point in the design space.

#### 6.3 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 specializable operations.

In particular, abstraction over type classes (similar to the proposal by Hughes [17] and used by L{"a}mmel and Peyton Jones [23]), and named type class instances [19] could help eliminate the boilerplate of reifying type classes as explicit dictionaries.

For example, in the definition of $gsum$, we defined both the type class $GSum$ and the type constructor $GumD$. We use the dictionary type in a number of ways. They allow alternate dictionaries (other than the standard one) to be provided to type representations, in the dynamic parameterization described in Section 5. Named instances for type classes would serve the same purpose. Also, explicit dictionaries allow the representation type to be parameterized by a type class and instance declarations to abstract over that type class. Type class abstraction would provide these capabilities.

With these capabilities, we would define the representation type so that type constructors include the context in their representation, not the explicit dictionaries. In otherwords, the definition of $RCon1$ would include the special cases in the context, instead of as an additional component.

```haskell
data Con1 c a = \forall l. CTup c l \Rightarrow Con1 (Emb l a)
data CT c a where
  CNil :: CT c Nil
  C :: a \Rightarrow CT c l \Rightarrow CT c (a * l)
cTup :: CT c l
```

With this facility, we may define polymorphic equality with no boilerplate. The code below completely defines the logic of the
operation. The only different from the non-extensible version (in Section 3) is the use of the \( R1 \) type and the recursive call through the type class in \( eqRL1 \).

```haskell
class Rep1 Eq a ⇒ Eq a where
t1 ≡ t2 = eqR1 rep1
eqR1 :: R1 Eq a → a → a → Bool
eqR1 lns1 = (≡)
eqR1 Char1 = (≡)
eqR1 (Data1 emb cons) = λx y → loop cons x y
where loop (Con1 emb : rest) x y =
case (from emb x, from emb y) →
  (Just kids1, Just kids2) →
eqRL1 ctup kids1 kids2
  (Nothing, Nothing) → loop rest
  (_ _) → False

eqRL1 :: CTup Eq l  → l → l → Bool
eqRL1 CNil Nil Nil = True
eqRL (C rtl) (p1 * t1) (p2 * t2) =
p1 ≡ p2 ∧ eqRL1 rtl t1 t2
```

Other language extensions that we plan to consider are mechanisms to support dynamic specialization 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-kinded 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. Representation types may also be implemented with a Church encoding [34]. However, in our view GADTs provide the best programming model for representation types: they support simple definitions of type-directed 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 require more uses of \( cast \) as there is no connection between arguments 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 \( Typeable \) type class is the foundation for the “Scrap your boilerplate” library [21, 22, 23]. This library includes a number of combinators for assembling type-indexed functions from smaller components. This style of programming is entirely compatible with RepLib—in fact we were able to port a module of traversal schemes (such as \( everywhere \)) to RepLib merely by renaming a single type class.

The idea of generically representing data constructors via isomorphisms (in Section 3) was first used by Generic Haskell and Derivable Type Classes [15], where data constructors were compiled to binary sums and products. It first saw specific use with representation 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. However, the specific isomorphism between data constructors and list of types is new to this paper. All of these isomorphisms provide similar expressive power—however, we think that manipulating type lists, either natively or with folds and maps, provides the most natural definition of type-indexed operations.

Derivable type classes [15] is closely related to the work described here. Like Generic Haskell, this approach treats datatypes as isomorphic to sums of products. However, as L¨ammel and Peyton Jones [23] point out, programming with datatypes in this manner 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 enable extensible type-indexed operations (Section 4) is new to this paper. It was inspired by the third “Scrap Your Boilerplate” paper of L¨ammel 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 functions 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 algorithm 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 version of Scrap your Boilerplate Reloaded, Hinze and L¨oh [14] describe 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 format. We intend to distribute and maintain this library, and accumulate new examples of type-indexed operations. (Several such operations are already included in the distribution.) Although this library 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.

References

A. Appendix

{-# OPTIONS -fglasgow-exts -fallow-undecidable-instances #-}
module RepLibLite where

import GHC.Base (unsafeCoerce#)

-- Simple representations (Section 3)

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

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

data DT = forall l. DT String (Rs l)
data Con a = forall l. Reps l => Con (Emb l a)
data Nil = Nil
data a :*: l = a :*: l
infixr 7 :*:

data Rs l where
  RNil :: Rs Nil
  (:^:) :: Rep a => R a -> Rs l -> Rs (a :*: l)
infixr 7 :^:

class Reps l where
  reps :: Rs l
instance Reps Nil where
  reps = RNil
instance (Rep a, Reps l) => Reps (a :*: l) where
  reps = rep :^: reps

class Rep a where rep :: R a

-- Casting

compR :: R a -> R b -> Bool
compR Int Int = True
compR Char Char = True
compR (Arrow t1 t2) (Arrow s1 s2) = compR t1 s1 && compR t2 s2
compR (Data rc1 _) (Data rc2 _) = compDT rc1 rc2
compR _ _ = False

compDT :: DT -> DT -> Bool
compDT (DT str1 rt1) (DT str2 rt2) = str1 == str2 && compRs rt1 rt2

compRs :: Rs t1 -> Rs t2 -> Bool
compRs RNil RNil = True
compRs (r1 :^: rt1) (r2 :^: rt2) = compR r1 r2 && compRs rt1 rt2

castR :: R a -> R b -> a -> Maybe b
castR (ra::R a) (rb::R b) =
  if compR ra rb
  then \(x::a) -> Just (unsafeCoerce# x::b)
  else \x -> Nothing

cast :: (Rep a, Rep b) => a -> Maybe b
cast (x :: a) :: Maybe b = castR (rep :: R a) (rep :: R b) x
gcastR :: \forall a b c. R a -> R b -> c a -> Maybe (c b)
gcastR ra rb = if compR ra rb
    then \(x :: c a) -> Just (unsafeCoerce# x :: c b)
    else \x -> Nothing

gcast :: \forall a b c. (Rep a, Rep b) => c a -> Maybe (c b)
gcast = gcastR (rep :: R a) (rep :: R b)

-------------------------------------------------------------------------------
-- Library operations
-------------------------------------------------------------------------------

data Val a = forall l. Reps l => Val (Emb l a) l
findCon :: [Con a] -> a -> Val a
findCon (Con emb : rest) x =
    case (from emb x) of
        Just kids -> Val emb kids
        Nothing -> findCon rest x

foldr_l :: (forall a. Rep a => a -> b -> b) -> b -> MTup R l -> l -> b
foldr_l f b MNil Nil = b
foldr_l f b (_ :+: rs) (a :*: l) = f a (foldr_l f b rs l )

foldl_l :: (forall a. Rep a => b -> a -> b) -> b -> MTup R l -> l -> b
foldl_l f b MNil Nil = b
foldl_l f b (_ :+: rs) (a :*: l) = foldl_l f (f b a) rs l

map_l :: (forall a. Rep a => a -> a) -> Rs l -> l -> l
map_l t RNil Nil = Nil
map_l t (_ :^: rs) (a :*: a1) = (t a :*: map_l t rs a1)

-------------------------------------------------------------------------------
-- Simple representations of built-in types
-------------------------------------------------------------------------------

instance Rep Int where rep = Int
instance Rep Char where rep = Char
instance (Rep a, Rep b) => Rep (a -> b) where rep = Arrow rep rep

-- Unit
rUnitEmb = Emb { to = \Nil -> (),
                   from = \() -> Just Nil }

instance Rep () where
    rep = Data (DT "()" RNil) [Con rUnitEmb]

-- Booleans
rTrueEmb = Emb { to = \Nil -> True,
                 from = \x -> if x then Just Nil else Nothing }

rFalseEmb = Emb { to = \Nil -> False,
                 from = \x -> if x then Nothing else Just Nil }

instance Rep Bool where
    rep = Data (DT "Bool" RNil) [Con rFalseEmb, Con rTrueEmb]

-- Pairs
rPairEmb :: Emb (a :*: b :*: Nil) (a,b)
rPairEmb =
    Emb { to = \( t1 :*: t2 :*: Nil) -> (t1,t2),
        from = \( t1 :*: t2 ) -> (t1,t2) }
from = \(a,b) \rightarrow \text{Just } (a :*: b :*: \text{Nil}) \}

instance \(\text{Rep } a, \text{Rep } b \Rightarrow \text{Rep } (a,b) \) where
rep = Data \((\text{DT }","\) \((\text{rep } : R a) :": (\text{rep } : R b) :": \text{RNil})\)
[ Con rPairEmb ]

-- Lists
rNilEmb :: Emb Nil [a]
rNilEmb =
Emb { to = \Nil \rightarrow [],
from = \x \rightarrow \text{case } x \text{ of}
(x:xs) \rightarrow \text{Nothing}
[] \rightarrow \text{Just } \text{Nil} }

rConsEmb :: Emb (a :*: [a] :*: \text{Nil}) [a]
rConsEmb =
Emb { to = \(\text{hd} :*: \text{tl} :*: \text{Nil}) \rightarrow (\text{hd} : \text{tl})
from = \x \rightarrow \text{case } x \text{ of}
(\text{hd} : \text{tl}) \rightarrow \text{Just } (\text{hd} :*: \text{tl} :*: \text{Nil})
[] \rightarrow \text{Nothing} }

instance \text{Rep } a \Rightarrow \text{Rep } [a] \) where
rep = Data \((\text{DT }"\"\) \((\text{rep } : R a) :": \text{RNil})\) [ Con rNilEmb, Con rConsEmb ]

---------------------------------------------------------------------------
-- Parameterized representations (Section 4)
---------------------------------------------------------------------------
data R1 ctx a where
  Int1 :: R1 ctx Int
  Char1 :: R1 ctx Char
  Arrow1 :: \(\text{Rep } a, \text{Rep } b \Rightarrow \text{ctx } a \rightarrow \text{ctx } b \rightarrow \text{R1 ctx } (a \rightarrow b)\)
  Data1 :: DT -> \([\text{Con1 ctx } a]\) -> \text{R1 ctx } a

data Con1 ctx a =
forall tl. Con1 (Emb tl a) (MTup (ctx) tl)
data MTup ctx a where
  MNil :: MTup ctx Nil
  (:+:) :: (Rep a) => ctx a -> MTup ctx l -> MTup ctx (a :*: l)
infixr 7 :+:

class Sat a where
dict :: a
class \text{Rep } a \Rightarrow \text{Rep1 ctx } a \) where
rep1 :: R1 ctx a

---------------------------------------------------------------------------
-- Library operations
---------------------------------------------------------------------------
data Val1 ctx a =
forall tl. Val1 (Emb tl a) l (MTup (ctx) l)

findCon1 :: [\text{Con1 ctx } a] \rightarrow a \rightarrow \text{Val1 ctx } a
findCon1 (Con1 emb rec : rest) x =
\text{case } (\text{from } emb x) \text{ of}
\text{Just } kids \rightarrow \text{Val1 emb kids rec}
\text{Nothing} \rightarrow \text{findCon1 rest } x

foldr_l1 :: (forall a. \text{Rep } a \Rightarrow \text{ctx } a \rightarrow a \rightarrow b \rightarrow b) \rightarrow b \rightarrow (\text{MTup } (\text{ctx}) l) \rightarrow l \rightarrow b
foldr_l1 f b MNil Nil = b
foldr_l1 f b (ca ++ c1) (a :*: l) = f ca a (foldr_l1 f b c1 l)

foldl_l1 :: (forall a. \text{Rep } a \Rightarrow \text{ctx } a \rightarrow b \rightarrow a \rightarrow b) \rightarrow b \rightarrow (\text{MTup } (\text{ctx}) l) \rightarrow l \rightarrow b
foldl_l1 f b MNil Nil = b
foldl_l1 f b (ca :+: cl) (a :*: l) = foldl_l1 f (f ca b a) cl l

map_l1 :: (forall a. Rep a => ctx a -> a -> a) -> (MTup (ctx) l) -> l -> l
map_l1 f MNil Nil = Nil
map_l1 f (ca :+: cl) (a :*: l) = (f ca a) :*: (map_l1 f cl l)

-- Parameterized representations of built-in types

instance Rep1 ctx Int where rep1 = Int1
instance Rep1 ctx Char where rep1 = Char1
instance (Rep a, Rep b, Sat (ctx a), Sat (ctx b)) => Rep1 ctx (a -> b) where
  rep1 = Arrow1 dict dict

-- Unit
instance Rep1 ctx () where
  rep1 = Data1 (DT "()" RNil) [Con1 rUnitEmb MNil]

-- Booleans
instance Rep1 ctx Bool where
  rep1 = Data1 (DT "Bool" RNil) [Con1 rFalseEmb MNil, Con1 rTrueEmb MNil]

-- Pairs
rPair1 :: forall a b ctx. (Rep a, Rep b) => ctx a -> ctx b -> R1 ctx (a,b)
rPair1 ca cb =
  case (rep :: R (a,b)) of
    Data dt _ -> Data1 dt [Con1 rPairEmb (ca :+: cb :+: MNil)]

instance (Rep a, Sat (ctx a), Rep b, Sat (ctx b)) => Rep1 ctx (a,b) where
  rep1 = rPair1 dict dict

-- Lists
rList1 :: forall a ctx. Rep a => ctx a -> ctx [a] -> R1 ctx [a]
rList1 ca cl = Data1 (DT "[]" ((rep :: R a) :-: RNil))
  [Con1 rNilEmb MNil, Con1 rConsEmb (ca :+: cl :+: MNil)]

instance (Rep a, Sat (ctx a), Sat (ctx [a])) => Rep1 ctx [a] where
  rep1 = rList1 dict dict