September 2006

RepLib: A library for derivable type classes

Stephanie Weirich
University of Pennsylvania, sweirich@cis.upenn.edu

Postprint version. Published in *Proceedings of the 2006 ACM SIGPLAN workshop on Haskell*, September 2006, pages 1-12.
Publisher URL: http://doi.acm.org/10.1145/1159842.1159844

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
Postprint version. Published in *Proceedings of the 2006 ACM SIGPLAN workshop on Haskell*, September 2006, pages 1-12.
Publisher URL: http://doi.acm.org/10.1145/1159842.1159844

RepLib: A Library for Derivable Type Classes

Stephanie Weirich
University of Pennsylvania
sweirich@cis.upenn.edu

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*]: Applicative (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 Eq type class defines the signature of polymorphic equality.

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

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

$$\begin{array}{l}\textbf{data } Tree\ a = Leaf\ a \mid Branch\ (Tree\ a)\ (Tree\ a)\\ \textbf{instance } Eq\ a \Rightarrow Eq\ (Tree\ a) \textbf{ where}\\ \quad (Leaf\ x1) \equiv (Leaf\ x2) = x1 \equiv x2\\ \quad (Branch\ t1\ t2) \equiv (Branch\ s1\ s2) = t1 \equiv s1 \wedge t2 \equiv s2\\ \quad \_ \equiv \_ = False\end{array}$$

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 $Eq\ T$.

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.

$$\begin{array}{l}\textbf{data } Tree\ a = Leaf\ a \mid Branch\ (Tree\ a)\ (Tree\ a)\\ \quad \textbf{deriving } (Eq)\end{array}$$

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 for Eq 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 Polytypic 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] 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).

Haskell'06 September 17, 2006, Portland, Oregon, USA.

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-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.

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.

$$
\begin{array}{l}
\textbf{data}\ R\ a\ \textbf{where} \\
\quad \mathit{Int} \quad :: R\ \mathit{Int} \\
\quad \mathit{Unit} \quad :: R\ () \\
\quad \mathit{Bool} \quad :: R\ \mathit{Bool} \\
\quad \mathit{Char} \quad :: R\ \mathit{Char} \\
\quad \mathit{Pair} \quad :: R\ a \to R\ b \to R\ (a, b) \\
\quad \mathit{Arrow} :: R\ a \to R\ b \to R\ (a \to b) \\
\quad \mathit{List} \quad :: R\ a \to R\ [a]
\end{array}
$$

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 Int requires that the type parameter be Int. This reasoning works in reverse, too. If we know that the type of a term is $R\ \mathit{Int}$, then we know that the term must either be the data constructor Int or $\bot$.

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 Ints that appear in a data structure. (In this paper, all functions whose first argument is a representation type end with a capital "R".)

$$
\begin{array}{l}
\mathit{gsumR} :: R\ a \to a \to \mathit{Int} \\
\mathit{gsumR}\ \mathit{Int}\ x = x \\
\mathit{gsumR}\ (\mathit{Pair}\ t1\ t2)\ (x1, x2) = \\
\quad \mathit{gsumR}\ t1\ x1 + \mathit{gsumR}\ t2\ x2 \\
\mathit{gsumR}\ (\mathit{List}\ t)\ l = \\
\quad \mathit{foldl}\ (\lambda s\ x \to (\mathit{gsumR}\ t\ x) + s)\ 0\ l \\
\mathit{gsumR}\ (\mathit{Arrow}\ t1\ t2)\ f = \mathit{error}\ \texttt{"urk!"} \\
\mathit{gsumR}\ \_\ x = 0
\end{array}
$$

[1] `http://www.cis.upenn.edu/~sweirich/RepLib`

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 Int branch because we know that in that branch the type a must be Int. So, even though the type signature says the branch should return an Int, it is acceptable to return the argument x of type a. In GADT terminology, the type a has been *refined* to Int. Furthermore, in the Pair branch, we know that the type a must be a tuple, so we may immediately destruct the argument. Likewise, in the 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 Ints, unit, booleans, characters, pairs and lists, when provided with the appropriate type representation for that argument. For example,

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

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

$$
\begin{array}{l}
\textbf{class}\ \mathit{GSum}\ a\ \textbf{where} \\
\quad \mathit{gsum} :: a \to \mathit{Int} \\
\textbf{instance}\ \mathit{GSum}\ \mathit{Int}\ \textbf{where} \\
\quad \mathit{gsum}\ x = x \\
\textbf{instance}\ \mathit{GSum}\ ()\ \textbf{where} \\
\quad \mathit{gsum}\ x = 0 \\
\textbf{instance}\ \mathit{GSum}\ \mathit{Bool}\ \textbf{where} \\
\quad \mathit{gsum}\ x = 0 \\
\textbf{instance}\ \mathit{GSum}\ \mathit{Char}\ \textbf{where} \\
\quad \mathit{gsum}\ x = 0 \\
\textbf{instance}\ (\mathit{GSum}\ a, \mathit{GSum}\ b) \Rightarrow \mathit{GSum}\ (a, b)\ \textbf{where} \\
\quad \mathit{gsum}\ (x1, x2) = \mathit{gsum}\ x1 + \mathit{gsum}\ x2 \\
\textbf{instance}\ (\mathit{GSum}\ a) \Rightarrow \mathit{GSum}\ [a]\ \textbf{where} \\
\quad \mathit{gsum}\ l = \mathit{foldl}\ (\lambda s\ x \to (\mathit{gsum}\ x) + s)\ 0\ l
\end{array}
$$

With this definition, only a little type information is required at the function call to disambiguate the Num class.

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

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 make it easier to *use*. However, by using type classes and representation types together, we can get the advantages of both definitions.

Consider a class Rep that includes all types that are *representable*.

$$\textbf{class}\ \mathit{Rep}\ a\ \textbf{where}\ \mathit{rep} :: R\ a$$

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

$$
\begin{array}{ll}
\textbf{instance}\ \mathit{Rep}\ \mathit{Int} & \textbf{where}\ \mathit{rep} = \mathit{Int} \\
\textbf{instance}\ \mathit{Rep}\ () & \textbf{where}\ \mathit{rep} = \mathit{Unit} \\
\textbf{instance}\ \mathit{Rep}\ \mathit{Bool} & \textbf{where}\ \mathit{rep} = \mathit{Bool}
\end{array}
$$

```
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
```

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)
instance GSum a ⇒ GSum [a]
```

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 $Pair$s and $List$s with the same code.

In this section, we describe a representation type that can generically 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 $\forall$ 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 heterogenous 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     = λ(a :*: Nil) → (Leaf a),
  from   = λx → case x of
             Leaf a → Just (a :*: Nil)
             _      → Nothing}
rBranchEmb ::
  Emb (Tree a :*: Tree a :*: Nil) (Tree a)
rBranchEmb = Emb
  { to   = λ(l :*: r :*: Nil) → (Branch l r),
    from = λx → 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.

```
data MTup c l where
  MNil :: MTup c Nil
  (:+:) :: Rep a ⇒ c a → MTup c l → MTup c (a :*: l)
infixr 7 :+:
```

Like the R type, the type index describes what type list the term represents. The (:+:) constructor includes $Rep\ a$ 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)
example2 = Int :+: Char :+: MNil
```

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 ∀ 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   :: ∀ a.Rep a ⇒ Con R (Tree a)
rLeaf   = Con rLeafEmb ((rep :: R a) :+: MNil)
rBranch :: ∀ a.Rep a ⇒ Con R (Tree a)
rBranch = Con rBranchEmb
                ((rep :: R (Tree a)) :+:
                 (rep :: R (Tree a)) :+: MNil)
```

The definition of Con described in this section contains only the minimum information required for representating 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 instrinsic to the datatype itself, including the name of the datatype and the representation of its parameters.

```
data DT = ∀ 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 ⇒ Rep (Tree a) where
  rep = Data (DT "Tree" ((rep :: 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) ⇒ a → Maybe b
```

and the related generalized cast

```
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.[2]

Also, displaying the representation of types such as $Tree\ Int$ or $Tree\ Bool$ requires both of the components of DT. The instance

[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.

```
data Val c a = ∀ l.Val (Emb l a) (MTup c l) l

findCon :: [Con c a] → a → Val c a
findCon (Con emb reps : rest) x = case (from emb x) of
  Just kids → Val emb reps kids
  Nothing → findCon rest x
findCon [] x = error "Invalid representation"

foldl_l :: (∀ a.Rep a ⇒ c a → b → a → b) → b
  → (MTup c l) → l → b
foldl_l f b MNil Nil = b
foldl_l f b (ra :+: rs) (a :*: l) = foldl_l f (f ra b a) rs l
map_l :: (∀ a.Rep a ⇒ c a → a → a)
  → MTup c l → l → l
map_l t MNil Nil = Nil
map_l t (r :+: rs) (a :*: a1) = (t r a :*: map_l t rs a1)
```

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) ++ " -> " ++ (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) = "  " ++ 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.

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 → a → 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 → gsumRl reps kids
            Nothing → findCon rest
        findCon [] = error "Invalid representation"
```

```
  -- Type structure-based definition
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 _ reps kids →
      foldl_l (λr a b → (gsumR r a) + b) 0 reps kids
gsumR _                  x = 0

  -- 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 ...
```

Figure 2. Generic Sum

```
gsumR _                  x = 0
gsumRl :: MTup R l → l → Int
gsumRl MNil Nil          = 0
gsumRl (r :+: rs) (a :*: l) = gsumR r a + gsumRl rs l
```

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* should never reach the [] case. If we have correctly represented the datatype, then one of the generic destructors will be able to decompose the type. This looping pattern appears often in type-indexed code, so it makes sense to factor it out. In Figure 1, we define the function *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.

Furthermore, once we have found the appropriate data constructor, the next step is often to iterate over the list of kids. Therefore, Figure 1 also contains the analogues of *foldl* and *map* for type lists.

With these operations, we can rewrite the *Data* branch for *gsumR* more succinctly as shown in Figure 2. (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.

The operations in Figure 1 make the definitions of some type-indexed functions very concise. For example, *deepSeq* below is an operation that fully evaluates its first argument. (The built-in Haskell operation *seq* only reduces its first argument to the outermost data constructor. This operation also recursively evaluates all of the kids of the data constructor too.)

```
deepSeqR :: R a → a → b → b
deepSeqR (Data dt cons) = λx →
  case (findCon cons x) of
    Val _ reps args →
      foldl_l (λra bb a → (deepSeqR ra a).bb) id reps args
deepSeqR _ = seq
```

Unlike 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.

```
deepSeq :: Rep a ⇒ a → b → b
deepSeq = deepSeqR rep
```

The operations *gsum* and *deepSeq* are examples of type-indexed *consumers*—functions that use type information to decompose an argument of that 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
  zero :: Rep a ⇒ a
  zero = zeroR rep

zeroR :: R a → a
zeroR Int = 0
zeroR Char = '0'
zeroR (Arrow z1 z2) = const (zeroR z2)
zeroR (Data dt (Con emb rec : rest)) =
  to emb (fromTup zeroR rec)

fromTup :: (∀ a.Rep a ⇒ c a → a) → MTup 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 maping the traversal across the subcomponents of a data constructor. Note that the annotation of the return type $a \to a$ 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 →
      case (findCon cons x) of
        Val emb reps kids →
          to emb (map_l (const 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 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, given the interesting case for increasing salaries.

```
increase :: Float → Company → Company
increase k = everywhere (mkT (incS k))

incS :: Float → Salary → Salary
incS k (S s) = S (s * (1 + k))
```

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 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 (Data _ cons) = λx y → loop cons x y
  where loop (Con emb reps : 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
      (_, _) → False

eqRl :: MTup R l → l → l → Bool
eqRl MNil Nil Nil = True
eqRl (r :+: rs) (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 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.

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

```
-- 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 (findCon cons x) of
    Val emb rec kids →
      foldl_l (λca a b → (gsumD ca b) + a) 0 rec kids
gsumR1 _ x = 0

-- Type 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
instance GSum Bool
  -- etc...

-- Special case for sets
instance GSum IntSet where
  gsum (IS l1) = gsum (nub l1)
```

Figure 3. Specializable Generic Sum

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.

```
newtype IntSet = IS [Int]


rSEmb :: Emb ([Int] :*: Nil) IntSet
rSEmb = Emb{ to   = λ(il :*: Nil) → IS il,
             from = λ(IS il) → Just (il :*: Nil) }
instance Rep IntSet where
  rep = Data (DT "IntSet" MNil)
             [Con rSEmb ((rep :: R [Int]) :+: MNil)]
```

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.

```
instance GSum IntSet where
  gsum (IS l1) = gsum (nub l1)
```

Unfortunately, with this instance, the behavior of generic sum for *IntSet*s 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).

```
gsum (IS [1, 1]) ≡ 1
gsum (Leaf (IS [1, 1])) ≡ 2
```

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 $R1$ below 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 $c\ a$. (Note that new definitions in this Section end with 1 to distinguish them from those of the previous section.)

```
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 -> [Con c a] -> R1 c a
```

As before, we create a (now multiparameter) type class to automatically 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
```

A function to create representation types must abstract the contexts that should be supplied for each of the kids. For example, the representation of $Tree$s 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) :+: MNil))
    [Con rLeafEmb (ca :+: MNil),
     Con rBranchEmb (ct :+: ct :+: 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 class Sat. This type class can be thought of as a "singleton" type class, the class of types that contain a single value. [3]

```
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 $Sat\ (c\ (Tree\ a))$ includes non-variables in the type.

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

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

```
instance Sat (c [Int]) => Rep1 c IntSet where
  rep1 = Data1 (DT "IntSet" MNil)
    [Con 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 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 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 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 $IntSet$s.

```
gsum (IS [1,1]) ≡ 1
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 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 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 making Eq and $Zero$ superclasses of Inc.

```
incR1 :: (Eq a, Zero a) => R1 IncD a -> a -> a
incR1 r a = if a ≡ zero
            then a
            else case r of
              Int1          -> a + 1
              Data1 _ cons ->
                case findCon cons a of
                  Val emb kids rec ->
                    to emb (map_l incD rec kids)
              _             -> a
```

```
class (Eq a, Zero a, Rep1 IncD a) => Inc a where
  inc :: a -> 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.

```
instance Rep T where
  rep = Data (DT "T" MNil) []
```

[3] In fact, the type $R\ t$ is also a singleton type for any t. Therefore, we could replace the class Rep with $Sat\ (R\ a)$.

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.

$$\textbf{instance}\ GSum\ T\ \textbf{where}\ gsum = gsumT$$

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

  $$\begin{array}{l}\textbf{data}\ MTup\ c\ l\ \textbf{where}\\ \quad MNil :: MTup\ c\ Nil\\ \quad (:+:)\ :: c\ a \to MTup\ c\ l \to MTup\ (a :*: l)\end{array}$$

  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

  $$\begin{array}{l}\textbf{data}\ MTup\ c\ l\ \textbf{where}\\ \quad MNil :: MTup\ c\ Nil\\ \quad (:+:)\ :: Rep1\ c\ a \Rightarrow\\ \qquad c\ a \to MTup\ c\ l \to MTup\ (a :*: l)\end{array}$$

  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

  $$\begin{array}{l}\textbf{data}\ MTup\ c\ l\ \textbf{where}\\ \quad MNil :: MTup\ c\ Nil\\ \quad (:+:)\ :: Rep\ a \Rightarrow c\ a \to MTup\ c\ l \to MTup\ (a :*: l)\\ \quad (:-:)\ :: Rep\ a \Rightarrow\\ \qquad R1\ c\ a \to MTup\ c\ l \to MTup\ (a :*: l)\end{array}$$

  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:

  $$\textbf{data}\ Sq\ a = L\ a \mid Br\ (Sq\ (a, a))$$

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

  $$\begin{array}{l}rSq1 :: \forall\ a\ c.\\ \quad Rep\ a \Rightarrow c\ a \to c\ (Sq\ (a, a)) \to R1\ c\ (Sq\ a)\\ rSq1\ c\ d = Data1\ (DT\ \texttt{"Sq"}\ ((rep :: R\ a) :+: MNil))\\ \quad [Con\ rLEmb \quad (c :+: MNil),\\ \quad\ Con\ rBREmb\ (d :+: MNil)]\end{array}$$

  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.)

  $$\begin{array}{l}\textbf{instance}\ (Rep\ a, Sat\ (c\ a), Sat\ (c\ (Sq\ (a, a)))) \Rightarrow\\ \quad Rep1\ c\ (Sq\ a)\ \textbf{where}\\ \qquad rep1 = rSq1\ dict\ dict\end{array}$$

  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.

  $$\begin{array}{l}rSq1 :: \forall\ a\ c.Rep\ a \Rightarrow c\ a \to R1\ c\ (Sq\ a)\\ rSq1\ d1 = Data1\ (DT\ \texttt{"Sq"}\ ((rep :: R\ a) :+: MNil))\\ \quad [Con\ rLEmb \quad (d1 :+: MNil),\\ \quad\ Con\ rBREmb\ (rSq1\ d1 :-: MNil)]\\ \textbf{instance}\ (Rep\ a, Sat\ (c\ a)) \Rightarrow Rep1\ c\ (Sq\ a)\ \textbf{where}\\ \quad rep1 = rSq1\ dict\end{array}$$

  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.

  $$\begin{array}{l}\textbf{instance}\ (Rep\ a, GSum\ a, GSum\ (Sq\ a)) \Rightarrow\\ \quad GSum\ (Sq\ a)\ \textbf{where}\\ \qquad gsum = gsumR1\ (rSq1\ dict\ dict)\end{array}$$

- Store the special cases in the context.

  $$\begin{array}{l}\textbf{data}\ MTup\ c\ a\ \textbf{where}\\ \quad MNil :: MTup\ c\ Nil\\ \quad (:+:) :: (Rep\ a, Sat\ (c\ a)) \Rightarrow\\ \qquad MTup\ c\ l \to MTup\ c\ (a :*: l)\end{array}$$

  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 mkT 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 ⇒ Lreduce b c where
  lreduce :: b → c → b
  lreduce = lreduceR1 rep1
lreduceR1 :: R1 (LreduceD b) c → b → c → b
lreduceR1 (Data1 rdt cons) b c =
  case (findCon cons c) of
    Val rcd rec args →
      foldl_l lreduceD b rec args
lreduceR1 _ b c = 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 (λb a → b ∧ p a) True
gand    :: (FL t) ⇒ t Bool → Bool
gand    = foldLeft (∧) 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 operations of arity one. Hinze [9] has noted that generierializing these operations to multi-arities 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
  MNil2   :: MTup2 c 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) = λx →
  let loop (Con2 rcd1 rcd2 ps : rest) =
    case from rcd1 x of
      Just a → to rcd2 (mapRL2 ps a)
      Nothing → loop rest
  in loop cons
mapR2 Int2 = id
mapR2 Char2 = id
mapRL2 :: MTup2 (→) l1 l2 → l1 → l2
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.

```
rTree2 :: ∀ a b c.c a b → c (Tree a) (Tree b)
   → R2 c (Tree a) (Tree b)
rTree2 a t = Data2 "Tree"
[Con2 rLeafEmb     rLeafEmb     (a :**: MNil2),
 Con2 rBranchEmb rBranchEmb (t :**: t :**: MNil2)]
```

The definition of *mapR2* and the representation of *Tree* derives an instance for the *Functor* constructor class.

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

```
  -- Universal operations
cast        :: (Rep a, Rep b)   ⇒ a → b
gcast       :: (Rep a, Rep b)   ⇒ c a → c b
deepSeq     :: Rep a            ⇒ a → b → b
subtrees    :: Rep a            ⇒ a → [a]
  -- Prelude operations
eqR1        :: R1 EqD a         → a → Bool
compareR1   :: R1 CompareD a    → a → a → Ordering
minBoundR1  :: R1 BoundedD a    → a
maxBoundR1  :: R1 BoundedD a    → a
showsPrecR1 :: R1 ShowD a       → ShowS
  -- Specializable operations
gsum        :: GSum a           ⇒ a → Int
zero        :: Zero a           ⇒ a
generate    :: Generate a       ⇒ Int → [a]
shrink      :: Shrink a         ⇒ a → [a]
lreduce     :: Lreduce a        ⇒ b → a → b
rreduce     :: Rreduce a        ⇒ a → b → b
```

Figure 4. Some type-indexed functions of RepLib

Representable forms	
Base types	Int
Parameterized base types	$\tau_1 \rightarrow \tau_2$, $IO\ \tau$
Newtypes	$\mathbf{newtype}\ T = MkT\ Int$
Uniform datatypes	$\mathbf{data}\ Nat = Z \mid S\ Nat$
Base-kind parameters	$Maybe$, $[\,]$
Abstract types, void types	$\mathbf{data}\ T$
Nested datatypes	$\mathbf{data}\ Sq\ a = L\ a \mid B\ (Sq\ (a, a))$
Simple GADTs	$\mathbf{data}\ T\ a\ \mathbf{where}\ I :: T\ Int$
Unrepresentable forms	
GADTs with existentials	$\mathbf{data}\ T\ a\ \mathbf{where}\ C :: b \rightarrow T\ Int$
Existential polymorphism	$\mathbf{data}\ T = \forall\ a.MkT\ a\ (a \rightarrow T)$
Universal polymorphism	$\mathbf{data}\ T = MkT\ (\forall\ a.a \rightarrow a)$
Higher-kinded parameters	$\mathbf{data}\ T\ c = MkT\ (c\ Int)$

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 simply by pairing a value with the representation of its type.

$$\mathbf{data}\ Dynamic = \forall\ a.Rep\ a \Rightarrow Dyn\ a$$

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 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 Rep type class, which can be done automatically. These operations include $cast$, $gcast$ 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 specializable version of $zero$. Other operations include a function that $generate$s 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 $R1$ 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 generically 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 specializable 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

gsum, 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:

```
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 boilerplate. The only difference from the non-extensible version (in Section 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
gsumR1 :: R1 GSum a → a → Int
gsumR1 Int            x = x
gsumR1 (Arrow1 _ _) x = error "urk"
gsumR1 (Data1 dt cons) x =
  case (findCon cons x) of
    Val emb rec kids →
      (fold_l1 (λ_ a b → gsum b + a) 0 rec kids
gsumR1 _              _ = 0
```

However, to define operations like *foldLeft* in the presence of context paramerization, we must be able to specify an alternate dictionary to be included in the representation. Named type class instances [19] would allow that behavior. 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 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 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 Lämmel 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 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 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]. Furthemore in the extended version of *Scrap your Boilerplate Reloaded*, Hinze and Löh [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. 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.

Acknowledgments

Thanks to the anonymous reviewers of the Haskell Workshop committee. This document was produced from literate Haskell source

with lhs2tex with help from Andres Löh. This work was partially supported by NSF grant CCF 0347289.

References

[1] Martín Abadi, Luca Cardelli, Benjamin Pierce, and Gordon Plotkin. Dynamic typing in a statically-typed language. *ACM Transactions on Programming Languages and Systems*, 13(2):237–268, April 1991.

[2] James Cheney and Ralf Hinze. First-class phantom types. CUCIS TR2003-1901, Cornell University, 2003.

[3] Dave Clarke, Ralf Hinze, Johan Jeuring, Andres Löh, and Jan de Wit. The Generic Haskell user's guide. Technical Report UU-CS-2001-26, Utrecht University, 2001.

[4] Karl Crary, Stephanie Weirich, and Greg Morrisett. Intensional polymorphism in type erasure semantics. *Journal of Functional Programming*, 12(6):567–600, November 2002.

[5] Bruno C. d. S. Oliveira and Jeremy Gibbons. TypeCase: A design pattern for type-indexed functions. In *Haskell Workshop*, Tallinn, Estonia, 2005.

[6] *DrIFT User Manual*, April 2006. Available at `http://repetae.net/~john/computer/haskell/DrIFT/`.

[7] The GHC Team. *The Glasgow Haskell Compiler User's Guide*, version 5.02 edition, 2002. Available at `http://www.haskell.org/ghc/`.

[8] Cordelia V. Hall, Kevin Hammond, Simon L. Peyton Jones, and Philip L. Wadler. Type classes in Haskell. *ACM Transactions on Programming Languages and Systems*, 18(2):109–138, March 1996.

[9] Ralf Hinze. Polytypic values possess polykinded types. In Roland Backhouse and J.N. Oliveira, editors, *Proceedings of the Fifth International Conference on Mathematics of Program Construction (MPC 2000)*, pages 2–27, Ponte de Lima, Portugal, July 2000.

[10] Ralf Hinze. Generics for the masses. In *International Conference on Functional Programming (ICFP)*, pages 236–243, September 2004.

[11] Ralf Hinze and Simon Peyton Jones. Derivable type classes. In Graham Hutton, editor, *Proceedings of the Fourth Haskell Workshop, Montreal, Canada, September 17, 2000*, volume 41.1 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science, August 2000.

[12] Ralf Hinze and Andres Löh. Scrap Your Boilerplate revolutions. In *8th International Conference on Mathematics of Program Construction, MPC 2006*, Kuressaare, Estonia, July 2006.

[13] Ralf Hinze, Andres Löh, and Bruno C. d. S. Oliveira. Scrap Your Boilerplate reloaded. In *Eighth International Symposium on Functional and Logic Programming, FLOPS 2006*, April 2005.

[14] Ralf Hinze, Andres Löh, and Bruno C. d. S. Oliveira. Scrap Your Boilerplate reloaded (extended version). Technical report, 2005. Available at `http://www.informatik.uni-bonn.de/~loeh/SYB0-TR.pdf`.

[15] Ralf Hinze and Simon Peyton Jones. Derivable type classes. In Graham Hutton, editor, *Proceedings of the 2000 Haskell Workshop, Montreal*, number NOTTCS-TR-00-1 in Technical Reports, September 2000.

[16] Stefan Holdermans, Johan Jeuring, Andres Löh, and Alexey Rodriguez. Generic views on data types. In *8th International Conference on Mathematics of Program Construction, MPC 2006*, Kuressaare, Estonia, July 2006.

[17] John Hughes. Restricted datatypes in Haskell. In *Haskell Workshop*, number UU-CS-1999-28, 1999.

[18] Patrick Jansson and Johan Jeuring. PolyP—A polytypic programming language extension. In *Twenty-Fourth ACMSIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 470–482, Paris, France, 1997.

[19] Wolfram Kahl and Jan Scheffczyk. Named instances for haskell type classes. In *Haskell Workshop 2001*, Firenze, Italy, September 2001.

[20] Oleg Kiselyov, Ralf Lämmel, and Keean Schupke. Strongly typed heterogeneous collections. In *Haskell '04: Proceedings of the ACM SIGPLAN workshop on Haskell*, pages 96–107, 2004.

[21] Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate: a practical approach to generic programming. In *ACM SIGPLAN International Workshop on Types in Language Design and Implementation (TLDI'03)*, pages 26–37, New Orleans, January 2003. ACM Press.

[22] Ralf Lämmel and Simon Peyton Jones. Scrap more boilerplate: reflection, zips, and generalised casts. In *ACM SIGPLAN International Conference on Functional Programming (ICFP'04)*, pages 244–255, Snowbird, Utah, September 2004. ACM.

[23] Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate with class: Extensible generic functions. In *ACM SIGPLAN International Conference on Functional Programming (ICFP'05)*, Tallin, Estonia, 2005.

[24] Andres Löh, Dave Clarke, and Johan Juering. Dependency-style Generic Haskell. In *ACM SIGPLAN International Conference on Functional Programming (ICFP)*, Uppsala, Sweden, 2003. To appear.

[25] Erik Meijer and Graham Hutton. Bananas in space: Extending fold and unfold to exponential types. In *FPCA95: Conference on Functional Programming Languages and Computer Architecture (FPLCA)*, pages 324–333, La Jolla, CA, June 1995.

[26] Martin Odersky and Konstantin Läufer. Putting type annotations to work. In *Conference Record of POPL '96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, St. Petersberg Beach, Florida*, pages 54–67, New York, N.Y., 1996.

[27] Bruno C.d.S. Oliveira, Ralf Hinze, and Andres Löh. Generics as a library. In *Seventh Symposium on Trends in Functional Programming, TFP 2006*, Nottingham, UK, April 2006.

[28] Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. Simple unification-based type inference for GADTs. In *International Conference on Functional Programming (ICFP)*, Portland, OR, USA, September 2006.

[29] Simon L. Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Mark Shields. Practical type inference for arbitrary-rank types. *Journal of Functional Programming*, 2006. To appear.

[30] T Sheard and SL Peyton Jones. Template meta-programming for Haskell. In Manuel Chakravarty, editor, *Proceedings of the 2002 Haskell Workshop, Pittsburgh*, October 2002.

[31] Mark Shields and Simon Peyton Jones. Lexically scoped type variables. Microsoft Research, 2002.

[32] Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad hoc. In *Sixteenth ACMSIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 60–76. ACM Press, 1989.

[33] Geoffrey Washburn and Stephanie Weirich. Good advice for type-directed programming: Aspect-oriented programming and extensible generic functions. In *Workshop on Generic Programming (WGP)*, Portland, OR, USA, September 2006.

[34] Stephanie Weirich. Encoding intensional type analysis. In D. Sands, editor, *10th European Symposium on Programming (ESOP)*, pages 92–106, Genova, Italy, April 2001.

[35] Stephanie Weirich. Higher-order intensional type analysis. In Daniel Le Métayer, editor, *11th European Symposium on Programming (ESOP)*, pages 98–114, Grenoble, France, April 2002.

[36] Stephanie Weirich. *Programming With Types*. PhD thesis, Cornell University, August 2002.

[37] Stephanie Weirich. Higher-order intensional type analysis in type erasure semantics, July 2003. At `http://www.cis.upenn.edu/~sweirich/papers/erasure/erasure-paper-july03.pdf`.

[38] Stephanie Weirich. Type-safe cast. *Journal of Functional Programming*, 14(6):681–695, November 2004.

[39] Stephanie Weirich. Type-safe run-time polytypic programming. *Journal of Functional Programming*, 2006. To appear.