— DRAFT [May 11, 2009] —

Instant Generics: Fast and Easy

Manuel M. T. Chakravarty Gabriel C. Ditu Roman Leshchinskiy
University of New South Wales, Australia
{chak,gabd,rl}@cse.unsw.edu.au

Abstract

This paper introduces a novel approach to datatype-generic programming based on type classes and type families. The approach favours simplicity as generic functions are based on Haskell's standard construct for ad-hoc polymorphism, namely type classes — hence, it integrates well with existing classes and facilitates overriding of generic behaviour with conventional class instances. Moreover, our approach is expressive, as we demonstrate by an evaluation along the criteria of a set of standard benchmark problems for generic programming in Haskell. Beyond these benchmarks, which only cover type-indexed functions, we fully support type-indexed data types as well as a novel generic view, which we call the *structural view*. Finally, our approach is designed to lead to highly efficient code with no or little overhead compared to hand-written datatype-specific code. Both, support for type-indexed data types and high performance were crucial in our principal application: a self-optimising, high-performance array library for data parallel programming in Haskell.

Keywords generic programming, type classes, type families,

type-indexed datatypes

1. Introduction

Datatype-generic programming boosts software reuse by enabling *generic functions* that can be used with many different datatypes. Datatype-generic functions are guided by the structure of a datatype — for example, a generic map or fold function may operate on both lists and trees by traversing the container structure to reach all list elements and tree leaves, respectively. Other datatype-generic functions operate uniformly on entire heterogeneous structures, such as testing for equality and serialisation.

However, there are many examples of datatype-generic functions that require a non-uniform treatment, where we want to override the generic default behaviour at specific types or at specific data constructors. For example, during serialisation, we might want to optimise the treatment of some large embedded structures or, given the abstract syntax tree of a compiler, we might want to compute the free variables of specific language constructs. In the latter example, by default, we want to traverse tree nodes while joining free-variable sets of subtrees. However, at variable usage and binding occurences, we need to be able to define exceptional cases to create singleton sets and to subtract from sets, respectively.

Finally, a whole category of applications of datatype-generic functions are operating on *datatype-generic datatypes* — also called *type-indexed types* [Hinze et al. 2004]. An example of these are collective operations on self-optimising type-indexed arrays that adapt their concrete representation to their element type; in fact, such arrays are at the heart of *Data Parallel Haskell* [Chakravarty et al. 2007, Peyton Jones et al. 2008] and they have been our original motivation for developing a new approach to generic programming in Haskell. No existing approach supported datatype-generic datatypes while at the same time meeting our requirements regarding extensibility and performance.

In this paper, we introduce a novel approach to datatype-generic programming that gracefully deals with all of the previously described forms of datatype-generic functions. It is based on type classes [Wadler and Blott 1989] and type families [Chakravarty et al. 2005b,a, Schrijvers et al. 2008]. We shall argue that our approach is easy to use. Haskell programmers are familiar with type classes for ad-hoc polymorphism. Existing classes can be reused, and overriding of generic behaviour is achieved with conventional class instances. Moreover, we describe the use of Template Haskell [Sheard and Jones 2002] to generate generic boilerplate code, such as structure representation types and conversion functions; alternatively, this code could be generated by an extension to the deriving mechanism of the Glasgow Haskell Compiler (GHC).

To evaluate the expressiveness of our approach, we adopt the comparative benchmark framework of Rodriguez et al. [2008]. In summary, our approach scores very well and satisfies almost all of the criteria for expressiveness. However, due to its use of type families, it is currently restricted to be used with GHC, and we

are still in the process of producing a comprehensive distribution. However, our framework is already in active use as part of the library backend of Data Parallel Haskell (which has been shipped as a pre-release with GHC since version 6.10.1).

Beyond the functionality assessed by the benchmark framework —which is restricted to datatype-generic functions— our approach also supports datatype-generic datatypes. These are datatypes whose representation is dependent on a type parameter. For example, datatype-generic containers are container types whose representation adapts to the structure of the type of the elements they contain. An intriguing example, which we will use for illustrative purposes, are Hinze's [2000] generic generalised tries.

Previous work on generic programming in Haskell identified a range of *generic views*, each with their own strengths and weaknesses. They were summarised and integrated into *Generic Haskell* by Holdermans et al. [2006]. The implementation of generic functions on container types, such as generic map and fold functions, requires a functorial representation of the traversed datatype, either by the *functorial (or lifted) sum-of-products view* [Hinze 2002] or by the *fixed-point view* [Jeuring and Jansson 1997], each with its own trade offs. We introduce a new view, which we call the *structural view*. It enables generic functions on container types —among

other functionality— while being a proper extension of the non-functorial sum-of-products view. Its advantage is extended functionality without the need for multiple views, but it increases the size of the generic boilerplate somewhat.

A distinct advantage of basing generic programming on type classes is that existing Haskell compilers put considerable effort into optimising the use of type classes. We pay special attention

ensuring that our generic type-class code optimises well and incurs as little runtime overhead as possible. Together with the need to support type-indexed datatypes, high-performance code was one of the original requirements for generic functions in the implementation of Data Parallel Haskell.

The specific contributions of this paper are the following:

- a novel approach to datatype-generic programming using type classes and type families (Section 2),
- a unique combination of expressiveness and extensibility (Section 3),
- a method to define type-indexed datatypes and associated generic functions (Section 4),
- high-performance generic code with minimal overhead over handwritten datatype-specific code (Section 5), and
- an evaluation of our approach based on the framework of Rodriguez et al. [2008] (Section 6).

There is plenty of related work on generic programming in Haskell. We refer to techniques that we build on throughout the text and provide a comprehensive summary of related work as part of the comparative evaluation in Section 6.

2. Simply generic

To convey an intuition of our approach, we shall start with the implementation of a number of well-known generic functions, which have repeatedly been used as benchmarks for generic programming [Hinze et al. 2007, Rodriguez et al. 2008].

We implement datatype-generic functions as methods of type classes, where the different instances of a class define the equations

of a function at different types. This is not unlike the proposals of Hinze and Peyton Jones [2001] (derivable type classes), of Weirich [2006] (RepLib), and of Oliveira et al. [2006a] (EMGM), but we use type families [Chakravarty et al. 2005b,a, Schrijvers et al. 2008] to gain extra flexibility and expressiveness. As in many other approaches to generic programming in Haskell, the key to a uniform, generic treatment of a wide range of data structures is the use of a fixed set of *structure representation types* [Hinze 1999, 2002]. In the following, we use the *sum-of-products view* with

```
data Unit    = Unit                 – singleton
data a :*: b = a :*: b              – product
data a :+: b = Inl a | Inr b        – sum
```

from `Data.Generics`. To represent the structure of an algebraic data type, we use the sum constructor `(:+:)` to separate alternatives and the product constructor `(:*:)` to combine multiple arguments of a single data constructor; finally, we use `Unit` for nullary constructors. In practice, it is convenient to also include constructor descriptors and record labels [c.f., Hinze and Peyton Jones 2001, Hinze et al. 2007], but we omit those in this paper to favour clarity of presentation.

On the type level, we associate a datatype with its structure representation by a synonym family `Repr`. For example, we have

```
type family Repr t
type instance Repr [a] = Unit :+: (a :*: [a])
```

On the value level, two functions


```
toRepr   :: Representable a => a -> Repr a
fromRepr :: Representable a => Repr a -> a
```

convert between datatypes and their representation. As the previous definition of `Repr [a]` indicates, we use a *shallow conversion* that converts one layer of a recursive structure at a time. For convenience, we combine the two conversion functions `toRepr` and `fromRepr` together with the type family `Repr` in a type class `Representable`.

2.1 Basic tree traversals

To illustrate the basic structure of our scheme for generic programming with type classes, let us consider the infamous binary encoder example [Hinze 1999, Hinze et al. 2007], which converts the value of a datatype into a stream of bits — we represent this as a class:

```
class Enc a where
  encode :: a -> [Bit]
```

To realise generic encoding, we first define the encoding of primitive types and representation types:

```
instance Enc Int where
  encode = encodeInt          – boring details omitted

instance Enc Unit where
  encode = const []

instance (Enc a, Enc b) => Enc (a :*: b) where
  encode (x :*: y) = encode x ++ encode y

instance (Enc a, Enc b) => Enc (a :+: b) where
  encode (Inl x) = 0 : encode x
  encode (Inr y) = 1 : encode y
```

On that basis, we can give a generic default implementation of `encode`:

```
dft_encode :: (Representable a, Enc (Repr a))
           => a -> [Bit]
dft_encode = encode . toRepr
```

As long as a datatype is `Representable`, we can encode it.

However, for recursive types —such as lists— we still miss an instance. Recall that our conversion to representation types is shallow; e.g., we had `Repr [a] = Unit :+: (a :*: [a])`. Hence, the represented type (here `[a]`) may appear in the representation. To cover that case, we need an appropriate instance:

```
instance Enc a => Enc [a] where
  encode = dft_encode
```

Consequently, we can use `encode` directly and need `dft_encode` only as a convenient shorthand to simplify instance declarations of types that use the generic default behaviour.

2.2 Extending existing classes

An advantage of generic programming based on type classes is that we can use them to extend existing classes, such as the standard class Eq for equality testing. The first step is to add instances for the representation types:

```
instance Eq Unit where
  Unit == Unit = True

instance (Eq a, Eq b) => Eq (a :*: b) where
  (x1 :*: y1) == (x2 :*: y2) = x1 == x2 && y1 == y2

instance (Eq a, Eq b) => Eq (a :+: b) where
  (Inl x) == (Inl y) = x == y
```

```
(Inr x) == (Inr y) = x == y
_       == _       = False
```

Now, assume we want to define equality for a user-defined type

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

First, we need to define a representation type (and the conversion functions, which are entirely standard and hence omitted):

```
type instance Repr (Tree a) =
                a :+: (Tree a :*: Tree a)
```

Then, we can simply use the default implementation implied by that representation:

```
instance Eq a => Eq (Tree a) where
  x == y = toRepr x == toRepr y
```

Another common example, where we might want to extend a standard class is Haskell's `show` function. This is straight forward to implement with our approach, but requires meta-data (such as constructor labels, fixity, and precedence) in representation types. This meta-data can be included in the standard way [Hinze et al. 2007], which is why we omit the details here.

2.3 Overriding generic definitions with ad-hoc definitions

It's a common requirement in generic programming that default generic functionality can be overridden at specific types, either because the generic repesentation does not apply (e.g., because the type is abstract) or because optimised or alternative functionality is required. In the case of the encode, we may like to optimise the encoding of lists by representing them by the length of the list

followed by the elements and thereby omitting the tags introduced by the sum constructor `(:+:)`. In our approach, much like other approaches to generic programming based on type classes [Hinze and Peyton Jones 2001, Weirich 2006, Oliveira et al. 2006a], we can easily use custom instances of the generic class:

```
instance Enc a => Enc [a] where
  encode l = encode (length l) ++ – encode list length
             concatMap encode l   – encode elements
```

Using the exact same mechanism, we can also implement benchmark applications, such as the widely cited "Paradise benchmark" [Lämmel and Peyton Jones 2003]. In conjunction with pattern matching, we can also conveniently realise exceptions to the generic default treatment of a datatype for just a single data constructor or any subset of the data constructors of a datatype. For example, given a binary tree with weights:

```
data Wtree a w = Wleaf  a
               | Wnode  (Wtree a w) (Wtree a w)
               | Weight (Wtree a w) w
```

We can special case the `Weight` constructor to remove it using an ad-hoc instance as follows:

```
instance (RmWeight a, RmWeight w)
  => RmWeight (Wtree a w) where
  rmWeight (Weight t _) = rmWeight t    – exception
  rmWeight t            = dft_rmWeight t – default
  where
    dft_rmWeight = fromRepr . rmWeight . toRepr
```

We omit the definition of the type representation for `Wtree` as well as the class definition for `RmWeight` and other instances as they

offer no new insights over our previous examples.

3. Extensibility

It is desirable to be able to extend frameworks for generic programming as well as to be able to extend generic functions implemented in a particular framework without the need to alter or even recompile existing code — this is related to Wadler's expression problem as discussed by Oliveira et al. [2006a]. Our approach to generic programming is based on type classes and type families, both of which are *open* language constructs that facilitate the addition of new class and type instances without recompilation. This property extends to our framework for generic programming, making it extensible along four axes (two of them strongly related) as described in the following.

3.1 Adding new views

So far, we have assumed the canonical sums-of-products view for structure representation types. However, as we will see in Section 4.1, and as discussed in detail by Holdermans et al. [2006], other views can be more suitable for some appliations.

The specific view —and hence, the set of structure representation types— is in fact entirely orthogonal to other aspects of our approach to generic programming. It is simply a matter of defining the type family `Repr` and the conversion functions `toRepr` and `fromRepr` appropriately. More generally, multiple views can be used side-by-side in a single application. They can even be mixed in the implementation of a single generic function on a datatype by datatype basis. In particular, given existing code using a particu-

lar view, we can add a new view and code using that new view by defining a new representation family `Repr2` and conversion functions `toRepr2` and `fromRepr2` together with appropriate instance declarations. This is possible without altering or recompiling existing code.

3.2 Extending an existing view

Not only can we add new views to existing code, but we can also extend an existing view by adding new representation types to that view and using them in new instances of `Repr`, `toRepr`, and `fromRepr`. In fact, this turns out to be a very useful feature in practice, as primitive types, such as `Char`, `Int`, `Float`, and so on belong to the representation types, too — the family `Repr` is the identity on these and other representation types.

It is not uncommon for the set of primitive types to grow. Be it that a new architecture enables the use of a different bitwidth for scalars (e.g., `Int64` and `Word64`) or that in a binding of a C library, we abstractly import new types. In fact, portable software usually has to handle different sets of primtive types depending on the architecture and operating system in use. Hence, a fixed set of representation types is, at the very least, rather inconvenient.

The openness of type families [Schrijvers et al. 2008] allows us to be flexible, to spread the definitions of representation types over multiple modules and to add them subsequently to a code base without altering the existing code. This turned out to be an important feature in our use of generics in Data Parallel Haskell. The generic array type of Data Parallel Haskell needs to be usable with any primitive type supported by GHC. The set of primitives types does not only vary between different versions and builds of the compiler, but these types are also defined in separate modules and occasionally in entirely separate libraries. Any closed, mono-

lithic approach to determining the representation types would be insufficient.

3.3 Subuniverses

As described in Section 2.1, a class associated with a generic function needs an, albeit simple instance for every datatype on which the generic function is supposed to operate, even if the generic

function should simply implement the generic default behaviour on that datatype. For example, we had

```
instance Enc a => Enc [a] where
  encode = dft_encode
```

This is somewhat tedious unless automated, but it has the advantage that we can easily define a specific generic function on only a subset of the entire universe of types that it may cover. The set of instance declarations precisely characterises the admissible types — and the compiler will statically detect any attempt to apply a generic function to a type outside of the subuniverse on which it is defined.

The subuniverse on which a generic function is defined is again not closed. It can be extended at any point by supplying additional instance declarations for types not covered previously.

3.4 Extending a generic function

Strongly related to the subuniverse covered by a given generic function, we can also extend a generic function to newly introduced types when this is required. Again, this works simply by means of an instance declaration at the new type. This instance declaration may simply use the generic default method, or use datatype-specific

code to override the default generic behaviour. As an example, consider the extension of equality with a user-defined `Tree` datatype in Section 2.2. The code we gave to define the datatype and extend the equality function could all be defined in a separate module — no existing code needs to be altered or re-compiled.

4. Generic views

A generic view [Holdermans et al. 2006] in our framework is given by a class

```
class Representable a where
  type Repr a                  – representation type of a
  toRepr   :: a -> Repr a      – shallow conversion...
  fromRepr :: Repr a -> a      – ...functions
```

comprising an associated type family `Repr` that maps a source datatype to its representation type under a particular generic view together with two conversion functions `toRepr` and `fromRepr`. The specific names of the class, the associated type, and the conversion functions is not fixed and we can have multiple co-existing representations. Generic code chooses a specific representation by using the appropriate class and its components. So far, we have introduced the class `Representable` as providing a sum-of-products view. In the next two subsection, 4.1 & 4.2, we will discuss two alternative views.

In fact, we can go a step further and instead of using a generic view merely as an ephemeral structure to define generic functions in a uniform manner, we can use it to actually represent a specific data structure in an optimised or more convenient manner. The result are datatype-generic datatypes as described in Subsection 4.3.

Finally, we may like to generate specific representation types and conversion functions (i.e., instances of the class `Representable`) automatically. In our use of generics for Data Parallel Haskell, we

use a special-purpose generator. However, for a more general use of our approach, we discuss automatic generation of representations in Subsection 4.4.

4.1 Functors in the lifted sums-of-products view

The implementation of generic functions on container types, such as generic map and fold functions, requires a functorial representation of the traversed datatype. A standard option, already proposed by Hinze [1999, 2002], is the lifted sums-of-products view, which uses the following representation types:


```
data I           r = I r
data K a         r = K a
data (f :**: g) r = f r :**: g r
data (f :++: g) r = Inl1 (f r) | Inr1 (g r)
```

We provide it in the form of an alternative representation class:

```
class Representable1 f where
  type Repr1 f :: * -> *
  toRepr1   :: f a -> Repr1 f a
  fromRepr1 :: Repr1 f a -> f a
```

ranging over functors of kind `* -> *`. In fact, much like for the `Typeable` class of Lämmel and Peyton Jones [2003], we may provide an entire family of classes `Representable`, `Representable1`, `Representable2`, and so on, to represent nullary, unary, binary, and so on type constructors.

We represent the constructor for lists with `Representable1` as

```
instance Representable1 [] where
  type Repr1 [] = K Unit :++: (I :**: [])
```

```
  toRepr1 []     = Inl1 (K Unit)
  toRepr1 (x:xs) = Inr1 (I x :**: xs)
  fromRepr1 (Inl1 (K Unit))      = []
  fromRepr1 (Inr1 (I x :**: xs)) = x:xs
```

and we represent binary trees (c.f., Section 2.2) as

```
instance Representable1 Tree where
  type Repr1 Tree = I :++: (Tree :**: Tree)
  toRepr1 (Leaf x)   = Inl1 (I x)
  toRepr1 (Node l r) = Inr1 (l :**: r)
  fromRepr1 (Inl1 (I x))      = Leaf x
  fromRepr1 (Inr1 (l :**: r)) = Node l r
```

On the basis of the lifted sums-of-products, we can now extend Haskell's standard `Functor` class to cover new datatypes, such as `Tree`, by using the generic default implementation:

```
instance Functor Tree where
  fmap = fromRepr1 . fmap h . toRepr1
```

As before, we might want to factorise the body of that method out into a toplevel function `dft_fmap` if we intend to use it in multiple instance declarations.

The behaviour of the generic default implementation is given by the `Functor` instances of the representation types; thusly,

```
instance Functor I where
  fmap h (I x) = I (h x)

instance Functor (K a) where
  fmap _ (K x) = K x

instance (Functor f, Functor g)
  => Functor (f :**: g) where
```

```
  fmap h (x :**: y) = fmap h x :**: fmap h y

instance (Functor f, Functor g)
  => Functor (f :++: g) where
  fmap h (Inl1 x) = Inl1 (fmap h x)
  fmap h (Inr1 y) = Inr1 (fmap h y)
```

4.2 The structural view

A disadvantage of the approach described in the previous subsection is that we need to maintain a family of representations to deal with type constructors of varying arity. An alternative is the *structural view*, where we expose additional detail about the structure of represented datatypes. In particular, we encode the occurence of variables and recursive uses of the datatypes explicitly in the

generic representation. To this end, we add two more types to the set of representation types:

```
data Var a = Var a
data Rec t = Rec t
```

This is another example, where we might want to extend the set of representations (c.f., Section 3.2).

Furthermore, we enrich the `Representable` instance for lists as follows:

```
instance Representable [a] where
  type Repr [a] = Unit :+: (Var a :*: Rec [a])

  toRepr (x:xs) = Inr (Var x :*: Rec xs)
  toRepr []     = Inl Unit
```

```
fromRepr (Inr (Var x :*: Rec xs)) = x : xs
fromRepr (Inl Unit)               = []
```

The data types `Var` and `Rec` are used as markers for variable subterms and recursive positions in the type term structure.

On the basis of this representation, we can implement —among other functions— a generic map without using constructor classes. More specifically, we parametrise over the element type of a container separate from the compound container type:

```
class Mappable t a b where
  type Rebind t a b
  mapit :: (a -> b) -> t -> Rebind t a b
```

Here `t` is the manifest type of the container (of kind `*`) before mapping and `a` and `b` are the element type before and after the mapping, respectively.

The instances for products and sums are structural, as usual, but the instance for `Var` is interesting:

```
instance Mappable (Var a) a b where
  type Rebind (Var a) a b = Var b
  mapit f (Var a) = Var (f a)
```

Here we replace one type variable by another, effectively tracking the type mapping `a -> b` of the mapped function `f`.

To deal with recursive occurences of the container type, we need to finally isolate the functor and apply it to the new type, while we recurse into the substructure at the value level:

```
instance Functor f => Mappable (Rec (f a)) a b where
  type Rebind (Rec (f a)) a b = Rec (f b)
  mapit f (Rec t) = Rec (mapp f t)
```

Now, we can instantiate the standard `Functor` class as before in Section 4.1.

This approach is related to the treatment of type constructors for a `Monad` class in the proposal for *parametric type classes* [Chen et al. 1992] and to the use of *traits classes* in C++.

4.3 Datatype-generic datatypes

As mentioned earlier an important feature of our approach to generic programming for its application in Data Parallel Haskell is the ability to define type-indexed data types, or datatype-generic datatypes [Hinze et al. 2004]. The highly-optimised sequential and multicore-parallel arrays of Data Parallel Haskell contain far too much detail too discuss here; however, the code is publicly available in the repository of `package dph` (c.f., Section 7).

Instead, we discuss an elegant data structure proposed by Hinze [2000]. It is a generalisation of tries to finite maps, where the map is specialised on the type of the keys indexing the map. We can express this using associated data families [Chakravarty et al. 2005b]:


```
class Representable k => GMapKey k where
  data GMap k :: * -> *
  empty       :: GMap k v
  lookup      :: k -> GMap k v -> Maybe v
  insert      :: k -> v -> GMap k v -> GMap k v
```

`GMap` is a datatype indexed by the key type `k`. Different instances of `GMap` may choose representation types for the finite maps in dependence on the key type.

First, we need to instantiate the class `GMapKey` for the representation types. We start with the basic types and uses an existing finite map library `Map.Map` for integer-keyed maps:

```
instance GMapKey Int where
```

```
  data GMap Int v         = GMapInt (Map.Map Int v)
  empty                   = GMapInt Map.empty
  lookup k (GMapInt m)    = Map.lookup k m
  insert k v (GMapInt m) = GMapInt (Map.insert k v m)

instance GMapKey Unit where
  data GMap Unit v           = GMapUnit (Maybe v)
  empty                      = GMapUnit Nothing
  lookup Unit (GMapUnit v)   = v
  insert Unit v (GMapUnit _) = GMapUnit $ Just v
```

Next, we cover products and sums, strictly following Hinze's [2000] proposal:

```
instance (GMapKey a, GMapKey b)
  => GMapKey (a :*: b) where
  data GMap (a :*: b) v
    = GMapPair (GMap a (GMap b v))
  empty          = GMapPair empty
  lookup (a :*: b) (GMapPair gm)
    = lookup a gm >>= lookup b
  insert (a :*: b) v (GMapPair gm)
    = GMapPair $ case lookup a gm of
        Nothing  -> insert a (insert b v empty) gm
Just gm2 -> insert a (insert b v gm2  ) gm

instance (GMapKey a, GMapKey b)
  => GMapKey (a :+: b) where
  data GMap (a :+: b) v
    = GMapEither (GMap a v) (GMap b v)
  empty                   = GMapEither empty empty
  lookup (Inl a) (GMapEither gm1  _gm2)
    = lookup a gm1
  lookup (Inr b) (GMapEither _gm1 gm2 )
```

```
    = lookup b gm2
  insert (Inl a) v (GMapEither gm1 gm2)
    = GMapEither (insert a v gm1) gm2
  insert (Inr a) v (GMapEither gm1 gm2)
    = GMapEither gm1 (insert a v gm2)
```

Given the generic definition of the datatype-generic type `GMap` and the corresponding datatype-generic functions `empty`, `lookup`, and `insert`, we can now simply use the generic default implementation for list-indexed finite maps:

```
instance GMapKey a => GMapKey [a] where
  newtype GMap [a] v = GMapList (GMap (Repr [a]) v)
  empty                          = GMapList empty
  lookup k (GMapList gm)    = lookup (toRepr k) gm
  insert k v (GMapList gm)
    = GMapList $ insert (toRepr k) v gm
```


4.4 Generating boilerplate

Although the purpose of a generic library is to eliminate boilerplate code, our framework itself requires a fair amout of boilerplate from the user who must provide the following components:

- instances of `Representable` for every user-defined data type that needs to interoperate with the framework,
- default implementations of generic functions which correctly convert to and from the representation types,
- instances which delegate to these default implementations for each supported combination of data types and generic functions.

Ultimately, our goal is to generate all of these automatically and we are optimistic that this is possible in the vast majority of cases. In the following, we briefly outline our approach and point out open problems.

Representation types and conversions `Representable` instances are trivially generated for arbitrary algebraic data types as they directly follow the structure of the type definition. Similar functionality is already implemented by several other libraries.

Default implementations In Section 2.1, we used `dft_encode` as an example of a default implementation for a generic function. Generally, such default implementations can be provided for arbitrary functions by appropriately converting to and from the generic representation. The question is: can default implementations such as `dft_encode` be generated automatically? Hinze and Peyton Jones [2001] show that this is indeed possible in most cases. They introduce the concept of *bidirectional mapping functions* which insert the necessary conversions exactly as required by our framework. We refer the reader to this previous work for an in-depth discussion of the technique. It should be pointed out, however, that some cases cannot be handled by this approach. Consider, for instance, the following class:

```
class C a where
  foo :: Functor f => f a -> f a
```

Here, the obvious default implementation relies on `fmap` to perform the conversions:

```
dft_foo :: (Functor f, Representable a, C (Repr a))
    => f a -> f a
dft_foo = fmap fromRepr . foo . fmap toRepr
```

We cannot reasonably expect `dft_foo` to be generated automatically as this would require the compiler to be aware of the semantics of `fmap`.

Boilerplate instances The final piece of boilerplate we would like to eliminate are instance declarations such as the `Enc [a]` instance from Section 2.1. Assuming that default implementations are available for all methods, the only real challenge lies in computing the context. In this particular example, it is rather trivial (`Enc a`) so let us consider a more complex one. Hinze and Peyton Jones [2001] give the following definition of generalised rose trees:

```
data GRose f a = GBranch a (f (GRose f a))
```

They argue that the obvious `Encode` instance:

```
instance (Enc a, Enc (f (GRose f a)))
  => Encode (GRose f a) where encode = dft_encode
```

is not feasible, mainly because it would cause the type checker to diverge. Fortunately, while this was indeed true when that paper was written, GHC now implements an extension to the type checking algorithm which supports the construction of *recursive dictionaries* and admits instances such as the one above [Lämmel and

Peyton Jones 2004]. This allows instance generation to be automated in the vast majority of cases as the required contexts can be derived directly from the data type declaration. Some unusual data types cannot be handled in this manner, however. An example are generalised nested rose trees:

```
newtype Comp f g a = Comp (f (g a))
data NGRose f a = NGRose a (f (NGRose (Comp f f) a))
```

We might try to use the same strategy as above to define an `Encode` instance for this type:

```
instance Enc (f (g a))
  => Enc (Comp f g a) where encode = dft_encode
instance (Enc a, Enc (f (NGRose (Comp f f) a)))
  => Enc (NGRose f a) where encode = dft_encode
```

Type checking diverges for these declarations, however. In fact, even with all type system extensions implemented by GHC, there simply is no meaningful `Enc (NGRose f a)` instance at all! This is a limitation of the Haskell type class system and is inherited rather than introduced by our approach. In general, we expect to be able to automatically generate an instance as long it can be expressed in Haskell at all and default implementations are available for all methods.

The strategies outlined in this section should handle most situations arising in real-world programs. In the short term, we intend to implement them as a suite of Template Haskell functions [Sheard and Jones 2002] which can be invoked by users to generate boilerplate code required for our framework. Ultimately, we envision a tighter integration with the Haskell syntax similar to the support for derivable type classes provided by GHC. The exact details are the subject of future research, however.

5. Performance

Ideally, a generic library should have little or no impact on performance as compared to hand-written code. Unfortunately, all too often the overhead is significant or even prohibitive. Rodriguez et al. [2008] identify slowdowns of between 2 and 80 for simple examples. This degree of inefficiency quickly becomes a show-stopper when performance is a concern.

In contrast, we do not expect the framework described in this paper to suffer from such problems in typical applications. In fact, its origins lie in the Data Parallel Haskell project where performance is of utmost importance and so far, we have been very satisfied with its efficiency. The only overhead introduced by the framework are conversions to and from the representation types. In the vast majority of cases, they can be eliminated automatically by a combination of inlining and standard code transformations. In the following, we describe this process in more detail before presenting a few preliminary benchmarks.

5.1 Fast by design

Let us return to the example from Section 2.2 where we provide a generic definition of equality on trees. If this implementation is to be used in real-world programs, its performance must be competitive with hand-written code. Fortunately, our approach does not obscure the structure of the code which allows an optimiser (in this case, GHC's simplifier [Peyton Jones and Santos 1998]) to automatically eliminate most or even all of the overhead using only standard transformations. In fact, in this particular example GHC ultimately generates the same code as for the hand-coded implementation of `Tree` equality.

To understand how the code is transformed, let us retrace the individual optimisation steps, starting with our original definition:

```
t == u = toRepr t == toRepr u
```


Since the representation type for `Tree` is a sum, `(==)` on the right-hand side of the definition refers to equality for `(:+:)`. The latter is inlined, giving (after desugaring):

```
t == u = case (toRepr t) of
           Inl x -> case (toRepr u) of
                      Inl y -> x == y
                      Inr _ -> False
           Inr x -> case (toRepr u) of
                      Inl _ -> False
                      Inr y -> x == y
```

Inlining `toRepr t`, we get:

```
t == u = case (case t of
                 Leaf x   -> Inl x
                 Node p q -> Inr (p :*: q)) of
           Inl x -> case (toRepr u) of
                      Inl y -> x == y
                      Inr _ -> False
           Inr x -> case (toRepr u) of
                      Inl _ -> False
                      Inr y -> x == y
```

The nested `case` can be simplified by a standard transformation (*case-of-case*), thus eliminating the overhead of converting a `Tree` to its representation type and immediately deconstructing the latter. Instead, the tree is inspected directly as it would be in hand-written code.

```
t == u = case t of
           Leaf x   -> case (toRepr u) of
                         Inl y -> x == y
                         Inr _ -> False
           Node p q -> case (toRepr u) of
                         Inl _ -> False
                         Inr y -> p :*: q == y
```

The two calls to `toRepr u` are resolved in the same way, by inlining `toRepr` and eliminating nested `case`.

```
t == u = case t of
           Leaf x   -> case u of
                         Leaf y   -> x == y
                         Node _ _ -> False
           Node p q -> case u of
                         Leaf _   -> False
                         Node r s
                           -> p :*: q == r :*: s
```

Finally, equality for `(:*:)` is inlined in the last branch and the resulting expression simplified. The end result is a definition which is exactly equivalent to what we would write by hand (modulo desugaring):

```
t == u = case t of
           Leaf x   -> case u of
                         Leaf y   -> x == y
                         Node _ _ -> False
           Node p q -> case u of
                         Leaf _   -> False
                         Node r s
                           -> p == r && q == s
```

To obtain this result, we have relied on a crucial property of our code: the conversion functions `toRepr` and `fromRepr` and the implementations of `(==)` for the representation types (in particular, `(:*:)` and `(:+:)`) are non-recursive. This is what allows the case-of-case transformation to completely remove the intermediate generic representation after inlining. In general, the compiler

	IG	hand-coded	SYB	Smash
geq	238	233	17096	997
rmWeights	541	408	93934	687
selectInt	1676	573	4842	2465
selectIntAcc	113	108	–	–

Figure 1. Benchmarks (runtime in ms)

should be able to eliminate all overhead introduced by our library if the following two conditions are satisfied.

1. The conversion functions can be inlined and are non-recursive.
2. The implementation of the generic function for the representation types can be inlined and is non-recursive.

We can expect many real-world applications to have these properties. In particular, the conversions only have to transform the "top layer" of a data structure and do *not* have to traverse it. Hence, as in the case of `Tree`, they are not recursive even if the data type they operate on is. Furthermore, most generic functions will have very simple implementations for the representation types — typically, they will just descend into the components and collect the results. This leads us to believe that our approach will have almost no overhead in most cases.

5.2 Benchmarks

Following Rodriguez et al. [2008], we have implemented three small benchmarks to investigate the efficiency of our framework:

- *geq* compares two binary trees for equality,
- *rmWeights* removes all weights from a weighted tree as described in Section 2.3,
- *selectInt* collects all integers from a weighted tree into a list.

The results are encouraging. Figure 1 compares the running times (in ms) of implementations using our library with hand-coded versions for sufficiently large data sizes. It also includes the results for implementations based on SYB [Lämmel and Peyton Jones 2003, 2004, 2005], which is readily available and widely used but also quite slow, and Smash [Kiselyov 2006], which has been identified as the fastest generic library by Rodriguez et al. [2008].

As explained in the previous section, generic equality introduces no overhead. The picture is different for `rmWeights` where our implementation is about 30% slower than the hand-coded version. This discrepancy is the result of the generic version doing strictly more work than the hand-coded implementation: while the latter simply removes all `Weight` nodes from the tree, the former also attempts to remove weights from the data elements stored in leaves. This introduces one superfluous method call for each leaf in the generic implementation. This behaviour can be simulated in the hand-coded implementation by extending it to take a function as an additional argument which is then applied to every integer stored in the tree. If invoked with the identity function, this version has exactly the same running time as the generic implementation.

Finally, for the `selectInt` the two implementations differ by a factor of 3. This large and quite surprising slowdown is due to a very subtle difference in how lists are constructed in the two

versions. In this benchmark, both the tree elements and the weights are of type `Int` and are collected in a single list. The hand-coded version is quite straight forward:

```
selectInt (Wleaf  n)   = [n]
selectInt (Wnode  x y) = selectInt x ++ selectInt y
selectInt (Weight x n) = n : selectInt x
```


Here, the last equation *prepends* the weight to the list yielded by the recursive call. In contrast to this, the generic version *appends* it which is a much less efficient operation. This difference is the sole reason for the large discrepancy in the performance.

However, neither of these two implementations will be used if efficiency is a concern. Rather, the algorithm would be optimised to use an accumulating parameter, thereby avoiding the need for list concatenation. For this version, which we call `selectIntAcc`, no perceptible overhead is introduced by using a generic implementation compared to the hand-coded one.

6. Related work and comparative evaluation

Increasing interest in generic programming has led to a steadily growing number of tools, frameworks and libraries for Haskell. These approaches can be divided into two categories: those that involve language extensions and/or preprocessors and those that use only supported language features (some language features may be supported through compiler extensions). Among those in the first category are Generic Haskell, Template Haskell, PolyP, Derivable Type Classes and DrIFT [see Hinze et al. 2007]. Since such approaches could in theory support arbitrary sets of features we limit the discussion in this section to the second category, to which our

approach belongs.

For these, Rodriguez et al. [2008] provide a comprehensive overview as well as giving an extensive benchmarking framework for assessing a generic programming library in Haskell. We use it as an independent validation tool and thus avoid the potential temptation of presenting examples or applications that work particularly well with our approach in favour of those that do not.

The benchmarking framework is built around a suite of tests designed to exercise various desirable qualities in the context of generic programming. The tests are applied uniformly to generic code implemented using the libraries being tested. We have not modified the tests in any way, but merely supplied our implementation as a new library plugin.

Below is a brief description of the main generic programming libraries for Haskell that we are aware off (Section 6.1), followed by an evaluation of our proposal together with a comparison with other libraries where applicable (Section 6.2).

6.1 Related work

In the following overview, we use the same library (nick)names as Rodriguez et al. [2008] to enable easy identification in the comparative evaluation section 6.2.

LIGD (Lightweight Implementation of Generics and Dynamics) [Cheney and Hinze 2002] encodes a product/sum representation of types on the value level. This allows generic functions to easily operate on type representations, effectively providing a *typecase* construct. A subsequent simplification of the representation type employs GADT's [Hinze et al. 2007].

SYB (Scrap Your Boilerplate) [Lämmel and Peyton Jones 2003, 2004] supports generic functions through the use of combinators. A type class encodes datatypes and provides primitive operations to consume or build values of a particular type. Generic functions

operate on all instances of the type class and are constructed by combining the primitive operations. Type-safe casting is used to support ad-hoc extensions.

SYB3 (SYB with class) [Lämmel and Peyton Jones 2005] represents an evolution of SYB that encodes generic functions as classes. The default behaviour can then be replaced by type-specific behaviour through class instances. These can be added in a modular way and at any time, thus eliminating the need for run-time type casts.

Spine (SYB with spine) [Hinze and Löh 2006] replaces the combinator-based approach of previous SYB iterations with a rep-

resentation type which is traversed by generic functions. The supported functionality is similar to that of SYB.

EMGM (Extensible and Modular Generics for the Masses) [Hinze 2006, Oliveira et al. 2006a] uses a type representation scheme augmented by a dispatch mechanism based on a type class which defines a separate method for each representation type. Generic functions are defined as types that are instances of that class. Extensibility is achieved through subclassing. This library is further extended in Oliveira et al. [2006b] with ad-hoc dispatchers for proper support of modular extensions.

RepLib [Weirich 2006] uses GADTs to define explicit structure representations, but combines those with type classes and, in particular, recursive dictionaries to free the programmer from explicitly manipulating the representations.

Smash (Smash your boilerplate) [Kiselyov 2006] is based on SYB and its various extensions. Generic functions in Smash are ordinary functions which define a generic traversal strategy and a list of special cases for particular data types. The implementation

is based on a static typecase operation.

Uniplate (Uniform boilerplate) [Mitchell and Runciman 2007] supports generic functions based on traversal combinators. Traversal functions are monomorphic and may be single- or multi-type. The latter are supported via multi-parameter type classes and allow ad-hoc behaviour for particular types.

PolyLib [Norell and Jansson 2004] is based on the earlier preprocessor language extension PolyP [Jeuring and Jansson 1997]. PolyLib supports a restricted type universe, single-parameter regular datatypes, and uses a pattern functor type class to encode type representations. The library provides a suite of generic operations over pattern functors.

Strafunski [Lämmel and Visser 2003] is aimed at language processing and centres around generic traversals and external plugins.

Compos [Bringert and Ranta 2008] is based on traversal combinators, in the spirit of Uniplate.

Finally, generic programming proposals exist that are based on other functional programming languages; Hinze et al. [2007] mention Clean, Charity and ML.

6.2 Comparative evaluation

The benchmark framework proposed by Rodriguez et al. [2008] defines a set of evaluation criteria for generic libraries. We address each of these below and provide comparisons to other approaches where appropriate. Figure 2 provides a summary of the supported features, in the format given in Rodriguez et al. [2008, p. 121].

Universe size: What types can be used by the generic code? Since our approach is based on type classes, generic code can use all types as long as appropriate instances can be defined in Haskell. This includes the vast majority of types used in real-world programs. Section 4.4 investigates this in more detail and also gives an example of a deeply nested type (`NGRose`) that cannot be sup-

ported by our framework. *Other libraries:* Most other approaches have trouble with `NGRose` and similar types, with only LIGD and Spine providing full support for them

Subuniverses: Is it possible to restrict the use of generic functions to a particular set of datatypes? Our generic functions can only operate on data types which have appropriate instance declarations. Thus, their use is *always* restricted to a particular set of datatypes: it is an opt-in model rather than opt-out. *Other libraries:* LIGD, SYB, SYB3, Spine and Uniplate do not support this feature.

First-class generic functions: Are generic functions first-class values? Our generic functions are polymorphic Haskell functions, so they are first-class. *Other libraries:* Uniplate does not support this criterion. EMGM and Smash require some special treatment.

Abstraction over type constructors: Can generic functions abstract over constructors? This feature is supported as shown in

Section 4. *Other libraries:* SYB, SYB3 and Uniplate do not support this feature. PolyLib and Spine only support constructors of kind `* -> *`.

Separate Compilation: Can a datatype, its type representation and a generic function be defined in different modules and used without modification or recompilation? This feature is naturally supported through our use of Haskell classes, which are extensible entities. *Other libraries:* The Spine library fails this criterion.

Ad-hoc definitions for datatypes: Can a generic function contain specific behaviour for a datatype? This is easily achieved by implementing the specific behaviour in the appropriate instance rather than use the generic implementation. Section 2.3 discusses this in more detail. *Other libraries:* LIGD and Spine require extensions to the representation type to support this feature.

Ad-hoc definitions for constructors: Can a generic function contain specific behaviour for a particular constructor? Yes; an example is provided in Section 2.3 where `rmWeight` only implements specialised functionality for `Weight` while handling all other constructors generically. *Other libraries:* The LIGD and Spine libraries require extensions to the representation type to support this. PolyLib supports this feature only for regular datatypes.

Extensibility: Can the programmer non-generically extend functionality in a different module? Yes, we extend functionality by defining new instances which can be done in different modules. *Other libraries:* This feature is not well supported by LIGD, Spine, SYB and Uniplate. PolyLib supports this feature only for regular datatypes.

Multiple arguments: Can functions be defined that have mutiple generic arguments? This feature is supported, generic equality is such an example. *Other libraries:* Uniplate does not support this feature, while SYB and Smash require undue programming effort.

Constructor names: Can the representation provide constructor names? Such a feature is needed to implement `show`-like functionality. This can be supported by extending the representation type to store additional information about the structure of types. We give here a simple extension to include constructor names, exemplified by the `Representable` instance for `BinTree`:

```
data Ctr a = Ctr a String

instance Representable (BinTree a) where
  type Repr (BinTree a) = (Ctr a) :+:
                          (Ctr (BinTree a :*: BinTree a))
  toRepr (Leaf a)  = Inl (Ctr a "Leaf")
  toRepr (Bin a b) = Inr (Ctr (a :*: b) "Bin")
  fromRepr (Inl (Ctr a _))          = Leaf a
```

```
fromRepr (Inr (Ctr (a :*: b) _)) = Bin a b
```

We wrap constructors in `Ctr` and attach the constructor name information. This is not enough for implementig the full `show` functionality correctly, but it serves as proof of concept and fulfills the requirements of the benchmark test. *Other libraries:* Uniplate does not give access to constructor names.

Consumers, transformers and producers: Can functions be written that consume, transform and produce generic types? We support all three operation classes. We have given examples of consumers and transformers; the `gfulltree` function in the benchmark produces test data generically based on the representation type. *Other libraries:* SYB, SYB3, Spine and Smash require different type representations for consumers and producers. Uniplate does not support producers.

Performance: Our approach offers performance similar to hand-written code, see Section 5 for discussion and performance benchmarks. Rodriguez et al. [2008] do not give specific peformance results but note that EMGM, Smash and Uniplate were fastest in their tests.

Criteria	Score
Universe Size	
Regular datatypes	●
Higher-kinded datatypes	●
Nested datatypes	●
Nested & higher-kinded datatypes	○
Mutually recursive datatypes	●
Subuniverses	●
First-class generic functions	●
Abstraction over type constructors	●
Separate compilation	●
Ad-hoc definitions for datatypes	●
Ad-hoc definitions for constructors	●
Extensibility	●
Multiple arguments	●
Constructor names	●
Consumers	●
Transformers	●
Producers	●
Performance	●
Portability	◐
Overhead of library use	
Automatic generation of representations	●
Number of structure representations	⋆
Work to instantiate a generic function	◐
Work to define a generic function	●
Practical aspects	○
Ease of learning and use	●

Key: ● Supported criterion
◐ Partially supported criterion
○ Unsupported criterion

Figure 2. Evaluation of our generic programming approach

Portability: What features are used that are not in the Haskell98 standard? We use associated type synonyms, a new feature that is quickly gaining popularity. We only rely on other type system extensions such as flexible contexts in certain more complex cases (cf. Section 4.4). *Other libraries:* PolyLib, SYB, SYB3, Spine, RepLib and Smash also score poorly on this criterion.

Overhead of library use: How much additional effort is required from the programmer to use the generic approach? Our scheme is simple and reasonably concise. We give a $\star$ for the number of structure representations in figure 2 because different representations are used according to the functionality required, see Section 3.1.

Practical aspects: Is there a current implementation and documentation? Also we have not yet released our framework, we intend to do so shortly. We hope that this paper provides a sufficient overview of our approach.

Ease of learning and use: This criterion is self-explanatory. We contend that our generic programming approach is very easy to learn and use, significantly more so than most, if not all, generics libraries we have seen. We use associated type synonyms in our representation type but their use is straight forward.

7. Conclusion

We presented a novel approach to datatype-generic programming in Haskell based on type classes and type families. Generic functions are defined as class methods, which makes them familiar first-class entities in Haskell, for which it is easy to define ad-hoc cases that

override the default generic functionality. As both type classes and type families are open language constructs that permit the addition of further instances in subsequent separately-compiled modules,

new generic views can be added or existing views and generic functions can be extended without altering or recompiling existing code.

We demonstrated that our approach to generic programming leads to very efficient code with minimal overhead over handwritten datatype-specific code with better performance than competing approaches — at least in our admittedly limited benchmarks. But our approach is not only efficient, it is also highly expressive as we showed by using a recent comparative benchmark framework for generic programming in Haskell.

Last but not least, we went beyond the conceptual boundaries of previous library-based approaches to generic programming in Haskell by supporting generic functions operating on datatype-generic datatypes (or type-indexed datatypes), such as generic generalised tries and self-optimising high-performance arrays. In addition, we introduced a new generic view that makes the entire structure of a represented datatype explicit.

Instant Generics are currently being used in the implementation of the library backend of Data Parallel Haskell (and widely distributed since GHC 6.10.1). They are used to implement a sequential and a multicore-parallel array library as part of `package dph`:

```
http://darcs.haskell.org/packages/dph/
```

The resulting code is, within the limits of GHC's code generator, competitive with handwritten C code for simple numerical kernels[1].

Moreover, we make the examples presented in this paper, and

several more, available at

`http://www.cse.unsw.edu.au/~chak/project/generics/`

To simplify the use of Instant Generics by other Haskell users, we are planning to develop the example code into a comprehensive library for generic programming in Haskell (independent of the code base in `package dph`).

Acknowledgements. We thank Gabriele Keller for lively discussions and feedback on a draft.

References

Björn Bringert and Aarne Ranta. A pattern for almost compositional functions. *Journal of Functional Programming*, 18:567–598, 2008. doi: 10.1017/S0956796808006898.

Manuel M. T. Chakravarty, Gabriele Keller, and Simon Peyton Jones. Associated type synonyms. In *ICFP '05: Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming*, pages 241–253. ACM Press, 2005a.

Manuel M. T. Chakravarty, Gabriele Keller, Simon Peyton Jones, and Simon Marlow. Associated types with class. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT Sysposium on Principles of Programming Languages*, pages 1–13. ACM Press, 2005b.

Manuel M. T. Chakravarty, Roman Leshchinskiy, Simon Peyton Jones, Gabriele Keller, and Simon Marlow. Data Parallel Haskell: a status report. In *DAMP 2007: Workshop on Declarative Aspects of Multicore Programming*. ACM Press, 2007.

Kung Chen, Paul Hudak, and Martin Odersky. Parametric type classes. In *ACM Conference on Lisp and Functional Programming*. ACM Press, 1992.

James Cheney and Ralf Hinze. A lightweight implementation of generics and dynamics. In *Haskell '02: Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell*, pages 90–104. ACM Press, 2002.

Ralf Hinze. A generic programming extension for haskell. In Erik Meijer, editor, *Proceedings of the Third Haskell Workshop*, number UU-CS-1999-28 in Technical Report. Universiteit Utrecht, 1999.

[1] `http://www.cse.unsw.edu.au/~chak/project/dph/`

Ralf Hinze. Generics for the masses. *Journal of Functional Programming*, 16(4&5):451–483, 2006.

Ralf Hinze. Polytypic values possess polykinded types. *Science of Computer Programming*, 43(2–3):129–159, 2002.

Ralf Hinze. Generalizing generalized tries. *Journal of Functional Programming*, 10(4):327–351, 2000.

Ralf Hinze and Andres Löh. "scrap your boilerplate" revolutions. In *Mathematics of Program Construction*, pages 180–208. Springer-Verlag, 2006.

Ralf Hinze and Simon Peyton Jones. Derivable type classes. In Graham Hutton, editor, *Proceedings of the 2000 ACM SIGPLAN Haskell Workshop*, volume 41.1 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science, 2001.

Ralf Hinze, Johan Jeuring, and Andres Löh. Type-indexed data types. *Science of Computer Programming*, 51:117–151, 2004.

Ralf Hinze, Johan Jeuring, and Andres Löh. Comparing approaches to generic programming in haskell. In Roland Backhouse, Jeremy Gibbons, Ralf Hinze, and Johan Jeuring, editors, *Lecture Notes of the Spring School on Datatype-Generic Programming 2006*, number 4719 in Lecture Notes in Computer Science, pages 72–149. Springer-Verlag, 2007.

Stefan Holdermans, Johan Jeuring, Andres Löh, and Alexey Rodriguez. Generic views on data types. In *Mathematics of Program Construction*, number 4014 in Lecture Notes in Computer Science, pages 209–234. Springer-Verlag, 2006.

Johan Jeuring and Patrik Jansson. PolyP — a polytypic programming language extension. In *Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 470–482. ACM Press, 1997.

Oleg Kiselyov. Smash your boilerplate without class and typeable. `http://article.gmane.org/gmane.comp.lang.haskell.general/14086`, 2006.

Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate: a practical approach to generic programming. In *Proceedings of the ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI 2003)*, pages 26–37, 2003.

Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate with class: extensible generic functions. In *ICFP '05: Proceedings of the tenth ACM SIGPLAN international conference on Functional programming*, pages 204–215. ACM Press, 2005.

Ralf Lämmel and Simon Peyton Jones. Scrap more boilerplate: reflection, zips, and generalised casts. In *ICFP '04: Proceedings of the ninth ACM SIGPLAN international conference on Functional programming*, pages 244–255. ACM Press, 2004.

Ralf Lämmel and Joost Visser. A Strafunski application letter. In *PADL '03: Proceedings of the 5th International Symposium on Practical Aspects of Declarative Languages*, pages 357–375, London, UK, 2003. Springer-Verlag. ISBN 3-540-00389-4.

Neil Mitchell and Colin Runciman. Uniform boilerplate and list processing. In *Haskell '07: Proceedings of the ACM SIGPLAN Workshop on Haskell workshop*, pages 49–60. ACM Press, 2007.

Ulf Norell and Patrik Jansson. Polytypic programming in Haskell. In *IFL'03: Implementation of Functional Languages*, volume 3145/2005, pages 168–184. Springer, 2004. ISBN 978-3-540-23727-3.

Bruno C. D. S. Oliveira, Ralf Hinze, and Andres Löh. Extensible and modular generics for the masses. In Henrik Nilsson, editor, *Trends in Functional Programming*, volume 7 of *Trends in Functional Programming*, pages 199–216. Intellect, 2006a.

Bruno C. D. S. Oliveira, Ralf Hinze, and Andres Löh. Generics as a library. In *Proceedings of the Seventh Symposium on Trends in Functional Programming*, 2006b.

Simon Peyton Jones and André L. M. Santos. A transformation-based optimiser for Haskell. *Sci. Comput. Program.*, 32(1-3):3–47, 1998. ISSN 0167-6423.

Simon Peyton Jones, Roman Leshchinskiy, Gabriele Keller, and Manuel M. T. Chakravarty. Harnessing the multicores: Nested data parallelism in Haskell. In Ramesh Hariharan, Madhavan Mukund, and V Vinay, editors, *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2008)*, Dagstuhl, Germany, 2008. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany. URL `http://drops.dagstuhl.de/opus/volltexte/2008/1769`.

Alexey Rodriguez, Johan Jeuring, Patrik Jansson, Alex Gerdes, Oleg Kiselyov, and Bruno C. d. S. Oliveira. Comparing libraries for generic programming in Haskell. In *Haskell '08: Proceedings of the first ACM SIGPLAN Symposium on Haskell*, pages 111–122. ACM Press, 2008.

Tom Schrijvers, Simon Peyton-Jones, Manuel M. T. Chakravarty, and Martin Sulzmann. Type checking with open type functions. In *Proceedings of ICFP 2008 : The 13th ACM SIGPLAN International Conference on Functional Programming*, pages 51–62. ACM Press, 2008.

Tim Sheard and Simon Peyton Jones. Template meta-programming for Haskell. In *Haskell '02: Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell*, pages 60–75. ACM Press, 2002.

P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad-hoc. In *Conference Record of the 16th Annual ACM Symposium on Principles of Programming Languages*, pages 60–76. ACM Press, 1989.

Stephanie Weirich. RepLib: a library for derivable type classes. In *Haskell '06: Proceedings of the 2006 ACM SIGPLAN Workshop on Haskell*, pages 1–12. ACM Press, 2006.