Draft version. *Rev* : 1169

Generic Programming for Indexed Datatypes

José Pedro Magalhães[1] Johan Jeuring[1,2]

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

{jpm,johanj}@cs.uu.nl

Abstract

An indexed datatype is a type that uses a parameter as a type-level tag; a typical example is the type of vectors, which are indexed over a type-level natural number encoding their length. Since the introduction of generalised algebraic datatypes, indexed datatypes have become commonplace in Haskell. Values of indexed datatypes are often more involved than values of plain datatypes, and programmers would benefit from having generic programs on indexed datatypes. However, no generic programming library adequately supports them, leaving programmers with the tedious task of writing repetitive code.

We show how to encode indexed datatypes in a generic programming library with type families and type-level representations in Haskell. Our approach can also be used in similar libraries, and is fully backwards-compatible. We show not only how to encode indexed datatypes generically, but also how to instantiate generic functions on indexed datatypes. Furthermore, all generic representations and instances are generated automatically, making life easier for users.

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

General Terms Languages

1. Introduction

Runtime errors are undesirable and annoying. Fortunately, the strong type system of Haskell eliminates many common programmer mistakes that lead to runtime errors, like unguarded casts. However, even in standard Haskell, runtime errors still occur often. A typical example is the error of calling *head* on an empty list.

Indexed datatypes, popular since the introduction of Generalized Algebraic Datatypes (GADTs, Peyton Jones et al. 2006), allow us to avoid calling *head* on an empty list and many other runtime errors by encoding further information at the type level. For instance, one can define a type of lists with a known length, and then define *head* in such a way that it only accepts lists of length greater than zero. This prevents the usual mistake by guaranteeing statically that *head* is never called on an empty list.

Datatype-generic programming (Gibbons 2007) is a programming technique that increases abstraction and reduces code duplication. Given the regular structure of inductive algebraic datatypes in Haskell, there is a small set of primitive operations upon which all datatypes are built. Datatype-generic programming exploits the isomorphism between a datatype and its representation, using only a small set of primitive types to provide functions that operate similarly on any datatype. Many functions depend only on the structure of the datatype, and can therefore be defined generically. Typical examples are equality, enumeration, conversion to and from strings or binary encodings, traversals, etc.

Unfortunately, datatype-generic programming and indexed datatypes do not mix well. The added complexity of the indices and associated type-level computations needs to be encoded in a generic fashion, and while this is standard in dependently-typed approaches to generic programming, we know of no Haskell approach dealing with indexed datatypes. In fact, even the standard deriving mechanism, which automatically generates instances for certain type classes, fails to work for GADTs, in general.

We argue that it is time to allow these two concepts to mix. Driven by an application that makes heavy use of both generic programming and indexed datatypes (Magalhães and De Haas 2011), we have developed an extension to a current generic programming library to support indexed datatypes. Our extension is conservative, in that it preserves all library functionality without requiring modifications to client code, and general, as it applies equally well to other libraries. Furthermore, we show that instantiating functions to indexed datatypes is not trivial, even in the non-generic case. In the context of datatype-generic programming, however, it is essential to be able to easily instantiate functions; otherwise, we lose the simplicity and reduced code duplication we seek. Therefore we show a way of automatically instantiating generic functions to indexed datatypes, which works for most types of generic functions.

The rest of this paper is organized as follows: we first introduce generic programming briefly in Section 2, and define indexed datatypes in Section 3. Section 4 deals with representing indexed datatypes generically, and Section 5 focuses on the problem of instantiation. Section 6 presents a general algorithm for automating the procedures described in the preceding two sections, and Section 7 deals with lifting a limitation of our encoding. Finally, we show related work in Section 8, present directions for future research in Section 9, and conclude in Section 10.

2. Generic programming with type families

In this paper we use a lightweight generic programming library using type-level representations with type families in a style similar to that first described by Chakravarty et al. (2009) and used by Van Noort et al. (2010) which we call `instant-generics` (the same name as its Hackage package).

2.1 Generic representation

The basic idea in datatype-generic programming is to devise a small number of primitive types that can be used to define a large number of derived types. If we can represent many complicated types using a small number of primitive types, we can also define functions that operate on the primitive types, and make these functions work on all types by converting to and from the primitive types appropriately. We call these primitive types the *representation types*, as they are used to represent all other types.

The `instant-generics` library uses the following representation types:

```
infixr 5 +
infixr 6 ×

data α + β = L α | R β
data α × β = α × β
data C γ α  = C α
data U      = U
data Var α  = Var α
data Rec α  = Rec α
```

As usual, we use sums to encode alternatives (different constructors), and products to encode multiple arguments to a constructor. Constructors are tagged with *C*, so that we can store meta-information such as constructor name, fixity, etc. After tagging with *C*, constructors without any arguments are encoded using the unit type *U*, and every argument to a constructor is further wrapped in a *Var* or *Rec* tag to indicate if the argument is a parameter of the type or a (potentially recursive) occurrence of a datatype.

To mediate between a value and its generic representation we use a type class:

```
class Representable α where
  type Rep α
  to   :: Rep α → α
  from :: α     → Rep α
```

A *Representable* type has an associated *Rep*resentation type, which is constructed using the types shown previously. We use a type family (Schrijvers et al. 2008) to encode the isomorphism between a type and its representation, together with conversion functions *to* and *from*.

As an example, we show the instantiation of the standard list datatype:

```
data List[]
instance Constructor List[] where conName _ = "[]"

data List:
instance Constructor List: where
  conName _ = ":"
  conFixity _ = Infix RightAssociative 5

instance Representable [α] where
  type Rep [α] = C List[] U
               + C List: (Var α × Rec [α])
  from []                      = L (C U)
  from (a : as)                = R (C (Var a × Rec as))
  to (L (C U))                 = []
```

$$to\ (R\ (C\ (Var\ a \times Rec\ as))) = (a : as)$$

The first lines deal with the constructor meta-information. Every constructor of the original datatype gives rise to an empty datatype, used as the first argument to *C* in the representation. The instances of the *Constructor* class provide the meta-information. Lists have two constructors: the empty case corresponds to a *L*eft

injection, and the cons case to a *R*ight injection. For the cons case we have two parameters, encoded using a product. The first one is a *Var*iable, and the second a *Rec*ursive occurrence of the list type.

2.2 Generic functions

Generic functions are defined by giving an instance for each representation type. We show two examples of generic functions: equality and enumeration.

2.2.1 Equality

We start the definition of generic equality with a type class and instances for the sum and product cases:

$$
\begin{array}{l}
\textbf{class}\ GEq'\ \alpha\ \textbf{where} \\
\quad geq' :: \alpha \to \alpha \to Bool \\
\\
\textbf{instance}\ (GEq'\ \alpha, GEq'\ \beta) \Rightarrow GEq'\ (\alpha + \beta)\ \textbf{where} \\
\quad geq'\ (L\ a)\ (L\ b) = geq'\ a\ b \\
\quad geq'\ (R\ a)\ (R\ b) = geq'\ a\ b
\end{array}
$$

$$\mathit{geq}'\ \_\quad\ \_\quad = \mathit{False}$$

$$\textbf{instance}\ (\mathit{GEq}'\ \alpha, \mathit{GEq}'\ \beta) \Rightarrow \mathit{GEq}'\ (\alpha \times \beta)\ \textbf{where}$$
$$\quad \mathit{geq}'\ (a \times b)\ (a' \times b') = \mathit{geq}'\ a\ a' \wedge \mathit{geq}'\ b\ b'$$

The instance for sums checks that both arguments are injected on the same side, and proceeds recursively. Products proceed recursively, requiring both arguments to be equal.

*U*nits are trivially equal, and *C*onstructors are equal if their arguments are equal:

$$\textbf{instance}\ \mathit{GEq}'\ U\ \textbf{where}$$
$$\quad \mathit{geq}'\ U\ U = \mathit{True}$$
$$\textbf{instance}\ (\mathit{GEq}'\ \alpha) \Rightarrow \mathit{GEq}'\ (C\ \gamma\ \alpha)\ \textbf{where}$$
$$\quad \mathit{geq}'\ (C\ a)\ (C\ b) = \mathit{geq}'\ a\ b$$

Finally, variables and recursive occurrences simply call another type-class for equality:

$$\textbf{instance}\ (\mathit{GEq}\ \alpha) \Rightarrow \mathit{GEq}'\ (\mathit{Var}\ \alpha)\ \textbf{where}$$
$$\quad \mathit{geq}'\ (\mathit{Var}\ a)\ (\mathit{Var}\ b) = \mathit{geq}\ a\ b$$
$$\textbf{instance}\ (\mathit{GEq}\ \alpha) \Rightarrow \mathit{GEq}'\ (\mathit{Rec}\ \alpha)\ \textbf{where}$$
$$\quad \mathit{geq}'\ (\mathit{Rec}\ a)\ (\mathit{Rec}\ b) = \mathit{geq}\ a\ b$$

This new type class *GEq* is used to aggregate all types we can compare for equality (be it generically or not):

$$\textbf{class}\ \mathit{GEq}\ \alpha\ \textbf{where}$$
$$\quad \mathit{geq} :: \alpha \rightarrow \alpha \rightarrow \mathit{Bool}$$

We can give ad-hoc instances for base types:

$$\textbf{instance}\ \mathit{GEq}\ \mathit{Char}\ \textbf{where}$$
$$\quad \mathit{geq} = (\equiv)$$

But what we mostly want is to be able to use the generic instances of the GEq' class. For this we need a default implementation, which defines how to apply generic equality to a representable type:

$$\begin{aligned} &geqDefault :: (Representable\ \alpha, GEq'\ (Rep\ \alpha)) \\ &\qquad\qquad \Rightarrow \alpha \rightarrow \alpha \rightarrow Bool \\ &geqDefault\ x\ y = geq'\ (from\ x)\ (from\ y) \end{aligned}$$

If a type is representable and its representation has an instance of GEq', we can compute generic equality by first converting from the original datatype and then calling geq' on the generic representation. All we are missing is a GEq instance for lists, now trivial:

$$\begin{aligned} &\mathbf{instance}\ (GEq\ \alpha) \Rightarrow GEq\ [\alpha]\ \mathbf{where} \\ &\quad geq = geqDefault \end{aligned}$$

2.2.2 Enumeration

Often generic functions are divided into three groups: consumers, transformers, and producers. Generic consumers, like equality, take a representable value and consume it into some constant type. Generic transformers, like *fmap*, take a representable type, transform it in some way, but return an element of the same type. Generic producers, like *read*, take some constant type and produce a representable type out of it. This distinction is important because often functions of the same group are defined in a similar style. Also, generic producers tend to be more problematic than consumers or transformers (Rodriguez Yakushev et al. 2008).

As an example of a generic producer, we show generic enumeration:

```
class GEnum′ α where
  genum′ :: [α]
instance GEnum′ U where
  genum′ = [U]
instance (GEnum α) ⇒ GEnum′ (Rec α) where
  genum′ = map Rec genum
instance (GEnum α) ⇒ GEnum′ (Var α) where
  genum′ = map Var genum
instance (GEnum′ α) ⇒ GEnum′ (C γ α) where
  genum′ = map C    genum′
```

We represent enumerations simply as lists of elements. There is only one unit, and the recursive and variable cases rely on another type-class, just as in the equality function. Constructors are generated by recursive invocation.

The most interesting cases are for sums, where we have a choice of what to generate, and products, where we have to combine all possible generated values:

$$\begin{array}{l}
\textbf{instance}\ (\mathit{GEnum}'\ \alpha, \mathit{GEnum}'\ \beta) \Rightarrow \mathit{GEnum}'\ (\alpha + \beta)\ \textbf{where} \\
\quad \mathit{genum}' = \mathit{map}\ \mathit{L}\ \mathit{genum}' \mathbin{+\!\!+} \mathit{map}\ \mathit{R}\ \mathit{genum}' \\
\textbf{instance}\ (\mathit{GEnum}'\ \alpha, \mathit{GEnum}'\ \beta) \Rightarrow \mathit{GEnum}'\ (\alpha \times \beta)\ \textbf{where} \\
\quad \mathit{genum}' = [\, x \times y \mid x \leftarrow \mathit{genum}', y \leftarrow \mathit{genum}' \,]
\end{array}$$

It is better to replace $(+\!\!+)$ with an operator that alternates the elements of the lists, and the list product with a diagonalization, thereby guaranteeing that every element of the datatype will eventually be generated. For the purposes of this paper those details are unimportant, so we take a simplistic implementation.

The default generic implementation simply maps the conversion function over the list of generated elements:

$$\begin{array}{l}
\mathit{genumDefault} :: (\mathit{Representable}\ \alpha, \mathit{GEnum}'\ (\mathit{Rep}\ \alpha)) \Rightarrow [\,\alpha\,] \\
\mathit{genumDefault} = \mathit{map}\ \mathit{to}\ \mathit{genum}'
\end{array}$$

We now define the top-level class *GEnum* and give ad-hoc instances for *Int* (simplified) and *Bool*, and a generic instance for lists:

$$\begin{array}{l}
\textbf{class}\ \mathit{GEnum}\ \alpha\ \textbf{where} \\
\quad \mathit{genum} :: [\,\alpha\,] \\
\textbf{instance}\ \mathit{GEnum}\ \mathit{Int}\quad \textbf{where} \\
\quad \mathit{genum} = [\,0\,..\,5\,] \\
\textbf{instance}\ \mathit{GEnum}\ \mathit{Bool}\ \textbf{where}
\end{array}$$

```
    genum = [True, False]
  instance GEnum α ⇒ GEnum [α] where
    genum = genumDefault
```

Finally, we can generate lists of integers, for instance: *take* 5 (*genum* :: [[*Int*]]) evaluates to [[ ], [0], [0, 0], [0, 0, 0], [0, 0, 0, 0]].

2.3 Similarities to other libraries

We chose to present `instant-generics` only for its simplicity; we know of at least three other libraries which use type families and type classes in a very similar way: `regular` (Van Noort et al. 2008), which further allows for mapping over container types; `multirec` (Rodriguez Yakushev et al. 2009), which further allows catamorphisms over mutually-recursive families of datatypes; and `generic-deriving` (Magalhães et al. 2010), which merges `instant-generics` and `regular`, due to be implemented in the Glasgow Haskell Compiler (GHC). These libraries work in a similar way to `instant-generics`, and the modification we describe in the next sections applies equally well to all of them.

3. Indexed datatypes

While the libraries described in the previous section already allow a wide range of datatypes to be handled in a generic fashion, they cannot deal with indexed datatypes. We call a datatype *indexed* if it has a type parameter that is not used as data (also called a *phantom* type parameter), but at least one of the datatype's constructors introduces type-level constraints on this type. The type of vectors, or size-constrained lists, is an example of such a datatype:

$$
\begin{array}{l}
\textbf{data}\ \mathit{Vec}\ \alpha\ \nu\ \textbf{where} \\
\quad \begin{array}{lll}
\mathit{Nil} & :: & \mathit{Vec}\ \alpha\ \mathit{Ze} \\
\mathit{Cons} & :: & \alpha \rightarrow \mathit{Vec}\ \alpha\ \nu \rightarrow \mathit{Vec}\ \alpha\ (\mathit{Su}\ \nu)
\end{array}
\end{array}
$$

The first parameter of *Vec*, α, is the type of the elements of the vector. In the GADT syntax above with type signatures for each constructor, we see that α appears as an argument to the *Cons* constructor; α is a regular type parameter. On the other hand, the second parameter of *Vec*, ν, does not appear as a direct argument to any constructor: it is only used to constrain the possible ways of building *Vec*s. We always instantiate ν with the following empty (uninhabited) datatypes:

$$
\begin{array}{l}
\textbf{data}\ \mathit{Ze} \\
\textbf{data}\ \mathit{Su}\ \nu \\
\\
\textbf{type}\ \mathit{0}_T = \mathit{Ze} \\
\textbf{type}\ \mathit{1}_T = \mathit{Su}\ \mathit{0}_T \\
\textbf{type}\ \mathit{2}_T = \mathit{Su}\ \mathit{1}_T
\end{array}
$$

A vector with two *Char*s, for instance, is represented as follows:

$$
\begin{array}{l}
\mathit{exampleVec} :: \mathit{Vec}\ \mathit{Char}\ \mathit{2}_T \\
\mathit{exampleVec} = \mathit{Cons}\ \texttt{'p'}\ (\mathit{Cons}\ \texttt{'q'}\ \mathit{Nil})
\end{array}
$$

Note that its type, $\mathit{Vec}\ \mathit{Char}\ \mathit{2}_T$, adequately encodes the length of the vector; giving any other type to *exampleVec*, such as $\mathit{Vec}\ \mathit{Char}\ \mathit{0}_T$ or *Vec Char Char*, would result in a type error.

Indexed types are easy to define as a GADT, and allow us to give more specific types to our functions. For instance, the type of vectors above allows us to avoid the usual `empty list` error when taking the first element of an empty list, since we can define a *head* function that does not accept empty vectors:

$$
\begin{array}{l}
head_{Vec} :: Vec\ \alpha\ (Su\ \nu) \rightarrow \alpha \\
head_{Vec}\ (Cons\ x\ \_) = x
\end{array}
$$

GHC correctly recognizes that it is not necessary to specify a case for $head_{Vec}\ Nil$, since that is guaranteed by the type-checker never to happen.

Indexed datatypes are also useful when specifying well-typed embedded languages:

$$
\begin{array}{lll}
\textbf{data}\ Term\ \alpha\ \textbf{where} & & \\
\quad Lit & :: Int & \rightarrow Term\ Int \\
\quad IsZero & :: Term\ Int & \rightarrow Term\ Bool
\end{array}
$$

$$
\begin{array}{lllll}
\mathit{Pair} & :: \mathit{Term}\ \alpha \to \mathit{Term}\ \beta & & & \to \mathit{Term}\ (\alpha, \beta) \\
\mathit{If} & :: \mathit{Term\ Bool} \to \mathit{Term}\ \alpha & \to \mathit{Term}\ \alpha & & \to \mathit{Term}\ \alpha
\end{array}
$$

The constructors of *Term* specify the types of the arguments they require and the type of term they build. We will use the datatypes *Vec* and *Term* as representative examples of indexed datatypes in the rest of this paper.

3.1 Type-level equalities and existential quantification

Indexed datatypes such as *Vec* and *Term* can be defined using only existential quantification and type-level equalities (Johann and Ghani 2008, Theorem 3). For example, GHC rewrites *Vec* to the following equivalent datatype:

$$
\begin{array}{lrl}
\textbf{data}\ \mathit{Vec}\ \alpha\ \nu = & \nu \sim \mathit{Ze} & \Rightarrow \mathit{Nil} \\
\mid & \forall \mu . \nu \sim \mathit{Su}\ \mu & \Rightarrow \mathit{Cons}\ \alpha\ (\mathit{Vec}\ \alpha\ \mu)
\end{array}
$$

The constructor *Nil* introduces the constraint that the type variable ν equals *Ze*; $\alpha \sim \beta$ is GHC's notation for type-level equality between types α and β. The *Cons* constructor requires ν to be a *Su* of something; this "something" is encoded by introducing an existentially-quantified variable μ, which stands for the length of the sublist, and restricting ν to be the sucessor of μ (in other words, one plus the length of the sublist).

This encoding of *Vec* is entirely equivalent to the one shown previously. While it may seem more complicated, it makes explicit what happens "behind the scenes" when using the *Nil* and *Cons* constructors: *Nil* can only be used if ν can be unified with *Ze*, and *Cons* introduces a new type variable μ, constrained to be the predecessor of ν. In the next section we will look at how to encode indexed datatypes generically; for this we need to know what kind of primitive operations we need to support. Looking at this definition of *Vec*, it is clear that we will need not only a way of encoding type equality constraints, but also introduction of new type variables.

3.2 Functions on indexed datatypes

The extra type safety gained by using indexed datatypes comes at a price: defining functions operating on these types can be harder. Consumer functions are not affected; we can easily define an evaluator for *Term*, for instance:

$$
\begin{array}{ll}
\mathit{eval} :: \mathit{Term}\ \alpha \to \alpha & \\
\mathit{eval}\ (\mathit{Lit}\ i) & = i \\
\mathit{eval}\ (\mathit{IsZero}\ t) & = \mathit{eval}\ t \equiv 0 \\
\mathit{eval}\ (\mathit{Pair}\ a\ b) & = (\mathit{eval}\ a, \mathit{eval}\ b) \\
\mathit{eval}\ (\mathit{If}\ p\ a\ b) & = \mathbf{if}\ \mathit{eval}\ p\ \mathbf{then}\ \mathit{eval}\ a\ \mathbf{else}\ \mathit{eval}\ b
\end{array}
$$

In fact, even GHC can automatically derive consumer functions on indexed datatype for us; the following *Show* instances work as expected:

$$
\begin{array}{l}
\mathbf{deriving\ instance}\ \mathit{Show}\ \alpha \Rightarrow \mathit{Show}\ (\mathit{Vec}\ \alpha\ \nu) \\
\mathbf{deriving\ instance}\ \mathit{Show}\ (\mathit{Term}\ \alpha)
\end{array}
$$

Things get more complicated when we look at producer functions. Let us try to define a function to enumerate values. For lists this is simple:

$$
\begin{array}{l}
\mathit{enum}_{[]} :: [\alpha] \to [[\alpha]] \\
\mathit{enum}_{[]}\ \mathit{ea} = [\,] : [x : \mathit{xs} \mid x \leftarrow \mathit{ea}, \mathit{xs} \leftarrow \mathit{enum}_{[]}\ \mathit{ea}]
\end{array}
$$

Given an enumeration of all possible element values, we generate all possible lists, starting with the empty list.[1] However, a similar version for *Vec* is rejected by the compiler:

[1] Once again we are not diagonalizing the elements from *ea* and $enum_{[]}\ ea$, but that is an orthogonal issue.

$$
\begin{aligned}
&enum_{Vec} :: [\alpha] \rightarrow [Vec\ \alpha\ \nu] \\
&enum_{Vec}\ ea = Nil : [Cons\ x\ xs \mid x \leftarrow ea, xs \leftarrow enum_{Vec}\ ea]
\end{aligned}
$$

GHC complains of being unable to match *Ze* with $Su\ \nu$, and rightfully so: we try to add *Nil*, of type $Vec\ \alpha\ Ze$, to a list containing *Cons*es, of type $Vec\ \alpha\ (Su\ \nu)$. To make this work we can use type classes:

$$
\begin{aligned}
&\textbf{instance}\ GEnum\ (Vec\ \alpha\ Ze)\ \textbf{where} \\
&\quad genum = [\,] \\
&\textbf{instance}\ \ (\ GEnum\ \alpha, GEnum\ (Vec\ \alpha\ \nu)) \\
&\qquad\quad \Rightarrow GEnum\ (Vec\ \alpha\ (Su\ \nu))\ \textbf{where} \\
&\quad genum = [Cons\ a\ t \mid a \leftarrow genum, t \leftarrow genum]
\end{aligned}
$$

In this way we can provide different types (and implementations) to the enumeration of empty and non-empty vectors.

Note that GHC (version 7.0.3) is not prepared to derive producer code for indexed datatypes. Trying to derive an instance $Read\ (Vec\ \alpha\ \nu)$ results in the generation of type-incorrect code. We show in Section 5.2 a way of identifying the necessary instances to be defined, which could also be used to fix this issue.

4. Handling indexing generically

As we have seen in the previous section, to handle indexed datatypes generically we need support for type equalities and quantification in the generic representation. We deal with the former in Section 4.1, and the latter in Section 4.2.

4.1 Type equalities

A general type equality $\alpha \sim \beta$ can be encoded in a simple GADT:

$$\begin{array}{l}\textbf{data}\ \alpha \simeq \beta\ \textbf{where}\\ \quad \mathit{Refl} :: \alpha \simeq \alpha\end{array}$$

We could add the $\simeq$ type to the representation types of `instant-generics`, and add type equalities as extra arguments to constructors. However, since the equalities are always introduced at the constructor level, and we have a representation type to encode constructors, we prefer to define a more general representation type for constructors which also introduces a type equality:

$$\begin{array}{l}\textbf{data}\ C_{Eq}\ \gamma\ \phi\ \psi\ \alpha\ \textbf{where}\\ \quad C_{Eq} :: \alpha \rightarrow C_{Eq}\ \gamma\ \phi\ \phi\ \alpha\end{array}$$

The new C_{Eq} type takes two extra parameters which are forced to unify by the C_{Eq} constructor. The old behavior of C can be recovered by instantiating the ϕ and ψ parameters to trivially equal types:

$$\textbf{type}\ C\ \gamma\ \alpha = C_{Eq}\ \gamma\ ()\ ()\ \alpha$$

Note that we can encode multiple equalities as a product of equalities. For example, a constructor which introduces the equality constraints $\alpha{\sim}\mathit{Int}$ and $\beta{\sim}\mathit{Char}$ would be encoded with a representation of type $C_{Eq}\ \gamma\ (\alpha \times \beta)\ (\mathit{Int} \times \mathit{Char})\ \delta$ (for suitable γ and δ).

4.1.1 Encoding types with equality constraints

At this stage we are ready to encode types with equality constraints that do not rely on existential quantification; the $\simeq$ type shown before is a good example:

$$
\begin{array}{l}
\textbf{instance}\ \mathit{Representable}\ (\alpha \simeq \beta)\ \textbf{where} \\
\quad \textbf{type}\ \mathit{Rep}\ (\alpha \simeq \beta) = C_{Eq}\ \simeq_{\mathit{Refl}}\ \alpha\ \beta\ U \\
\quad \mathit{from}\ \mathit{Refl} \qquad\quad = C_{Eq}\ U \\
\quad \mathit{to}\quad\ (C_{Eq}\ U) = \mathit{Refl}
\end{array}
$$

The type equality introduced by the *Refl* constructor maps directly to the equality introduced by C_{Eq}, and vice-versa. As *Refl* has no arguments, we encode it with the unit representation type U. The auxiliary datatype $\simeq_{Refl}$, which we omit, is used to encode constructor information about $Refl$, as usual in `instant-generics`.

4.1.2 Generic functions over equality constraints

We need to provide instances for the new C_{Eq} representation type for each generic function. The instances for the equality and enumeration functions of Section 2.2 are:

$$
\begin{array}{l}
\textbf{instance}\ (GEq'\ \alpha) \Rightarrow GEq'\ (C_{Eq}\ \gamma\ \phi\ \psi\ \alpha)\ \textbf{where} \\
\quad geq'\ (C_{Eq}\ a)\ (C_{Eq}\ b) = geq'\ a\ b \\
\\
\textbf{instance}\ (GEnum'\ \alpha) \Rightarrow GEnum'\ (C_{Eq}\ \gamma\ \phi\ \phi\ \alpha)\ \textbf{where} \\
\quad genum' = map\ C_{Eq}\ genum' \\
\textbf{instance}\qquad\qquad\qquad\quad GEnum'\ (C_{Eq}\ \gamma\ \phi\ \psi\ \alpha)\ \textbf{where} \\
\quad genum' = [\,]
\end{array}
$$

Generic consumers, such as equality, are generally not affected by the equality constraints. We do not bother requiring that ϕ and ψ are equal because there is no way to build a value which does not obey that restriction. Generic producers are somewhat trickier, because we are now trying to build a generic representation, and thus must take care not to build impossible cases. For generic enumeration, we proceed normally in case the types unify, and return the empty enumeration in case the types are different. Note that these two instances overlap, but remain decidable.

4.2 Existentially-quantified indices

Recall the shape of the *Cons* constructor of the *Vec* datatype:

$$\forall \mu . \nu \sim Su\ \mu \Rightarrow Cons\ \alpha\ (Vec\ \alpha\ \mu)$$

We need to be able to introduce new type variables in the type representation. A first idea would be something like:

$$\textbf{type}\ Rep\ (Vec\ \alpha\ \nu) = \forall \mu . C_{Eq}\ Vec_{Cons}\ \nu\ (Su\ \mu) \ldots$$

This however is not accepted by GHC, as the right-hand side of a type family instance cannot contain quantifiers. This restriction is well justified, as allowing this would lead to higher-order unification problems.

Another attempt would be to encode representations as data families instead of type families, so that we can use regular existential quantification:

$$\begin{aligned}&\textbf{data instance}\ Rep\ (Vec\ \alpha\ \nu) = \\ &\quad \forall \mu . Rep_{Vec}\ (C_{Eq}\ Vec_{Cons}\ \nu\ (Su\ \mu) \ldots)\end{aligned}$$

However, we do not want to use data families to encode the generic representation, as these introduce a new constructor per datatype, thereby effectively precluding a generic treatment of all types.

4.2.1 Faking existentials

Since the conventional approaches do not work, we turn to some more unconventional approaches. All we have is an index type variable ν, and we need to generate existentially-quantified variables that are constrained by ν. We know that we can use type families to create new types from existing types, so let us try that. We introduce a type family

$$\textbf{type family}\ X\ \nu$$

and we will use $X\ \nu$ where the original type uses μ. We can now write a generic representation for *Vec*:

$$\begin{aligned}
&\textbf{instance}\ \mathit{Representable}\ (\mathit{Vec}\ \alpha\ \nu)\ \textbf{where}\\
&\quad\textbf{type}\ \mathit{Rep}\ (\mathit{Vec}\ \alpha\ \nu) = C_{Eq}\ \mathit{Vec}_{Nil}\ \ \nu\ \mathit{Ze}\ U\\
&\qquad\qquad\qquad\qquad + C_{Eq}\ \mathit{Vec}_{Cons}\ \nu\ (\mathit{Su}\ (X\ \nu))\\
&\qquad\qquad\qquad\qquad\qquad (\mathit{Var}\ \alpha \times \mathit{Rec}\ (\mathit{Vec}\ \alpha\ (X\ \nu)))
\end{aligned}$$

Draft version

$$\begin{aligned}
&\mathit{from}\ \mathit{Nil} &&= L\ (C_{Eq}\ U)\\
&\mathit{from}\ (\mathit{Cons}\ h\ t) &&= R\ (C_{Eq}\ (\mathit{Var}\ h \times \mathit{Rec}\ t))\\
&\mathit{to}\ (L\ (C_{Eq}\ U)) &&= \mathit{Nil}\\
&\mathit{to}\ (R\ (C_{Eq}\ (\mathit{Var}\ h \times \mathit{Rec}\ t))) &&= \mathit{Cons}\ h\ t
\end{aligned}$$

This is a good start, but we are not done yet, as GHC refuses to accept the code above with the following error:

```
Could not deduce (m ~ X (Su m))
from the context (n ~ Su m)
bound by a pattern with constructor
  Cons :: forall a n. a -> Vec a n
                        -> Vec a (Su n),
in an equation for ‘from’
```

What does this mean? GHC is trying to unify μ with $X\ (\mathit{Su}\ \mu)$, when it only knows that $\nu{\sim}\mathit{Su}\ \mu$. The equality $\nu{\sim}\mathit{Su}\ \mu$ comes from the pattern match on *Cons*, but why is it trying to unify μ with $X\ (\mathit{Su}\ \mu)$? Well, on the right-hand side we use C_{Eq} with type

$C_{Eq}\ Vec_{Cons}\ \nu\ (Su\ (X\ \nu))\ldots$, so GHC tries to prove the equality $\nu{\sim}Su\ (X\ \nu)$. In trying to do so, it replaces ν by $Su\ \mu$, which leaves $Su\ \mu{\sim}Su\ (X\ (Su\ \mu))$, which is implied by $\mu{\sim}X\ (Su\ \mu)$, but GHC cannot find a proof of the latter equality.

This is unsurprising, since indeed there is no such proof. Fortunately we can supply it by giving an appropriate type instance:

$$\textbf{type instance}\ X\ (Su\ \mu) = \mu$$

We call instances such as the one above "mobility rules", as they allow the index to "move" through indexing type constructors (such as *Su*) and *X*. Adding the type instance above makes the *Representable* instance for *Vec* compile correctly. Note also how *X* behaves much like an extraction function, getting the parameter of *Su*.

Representation for Term. The *Term* datatype (shown in Section 3) can be represented generically using the same technique. First let us write *Term* with explicit quantification and type equalities:

$$\begin{array}{llll}
\multicolumn{4}{l}{\textbf{data}\ Term\ \alpha =} \\
 & \alpha{\sim}Int & \Rightarrow Lit & Int \\
\mid & \alpha{\sim}Bool & \Rightarrow IsZero & (Term\ Int) \\
\mid & \forall\beta\ \gamma.\alpha{\sim}(\beta,\gamma) & \Rightarrow Pair & (Term\ \beta)\ (Term\ \gamma) \\
\mid & & If & (Term\ Bool)\ (Term\ \alpha)\ (Term\ \alpha)
\end{array}$$

We see that the *Lit* and *IsZero* constructors introduce type equalities, and the *Pair* constructor abstracts from two variables. This means we need two type families:

$$\textbf{type family}\ X_1\ \alpha$$
$$\textbf{type family}\ X_2\ \alpha$$

Since this strategy could require introducing potentially many type families, we use a single type family instead, parametrized over two other arguments:

$$\textbf{type family}\ X\ \gamma\ \iota\ \alpha$$

We instantiate the γ parameter to the constructor representation type, ι to a type-level natural indicating the index of the introduced variable, and α to the datatype index itself.

The representation for *Term* becomes:

$$\begin{array}{l}
\textbf{type}\ Rep_{Term}\ \alpha = \\
\quad\ \ C_{Eq}\ Term_{Lit}\ \ \ \ \alpha\ Int\ \ \ (Rec\ Int) \\
\quad + C_{Eq}\ Term_{IsZero}\ \alpha\ Bool\ (Rec\ (Term\ Int)) \\
\quad + C_{Eq}\ Term_{Pair}\ \ \ \alpha\ (X\ Term_{Pair}\ 0_T\ \alpha, X\ Term_{Pair}\ 1_T\ \alpha) \\
\qquad\qquad (\ \ \ Rec\ (Term\ (X\ Term_{Pair}\ 0_T\ \alpha)) \\
\qquad\qquad \times Rec\ (Term\ (X\ Term_{Pair}\ 1_T\ \alpha)))
\end{array}$$

$$\begin{aligned} + C \quad & \mathit{Term}_{\mathit{If}} \\ & (\mathit{Rec}\ (\mathit{Term\ Bool}) \times \mathit{Rec}\ (\mathit{Term}\ \alpha) \times \mathit{Rec}\ (\mathit{Term}\ \alpha)) \end{aligned}$$

We show only the representation type $\mathit{Rep}_{\mathit{Term}}$, as the *from* and *to* functions are trivial. The mobility rules are induced by the equality constraint of the *Pair* constructor:

$$\textbf{type instance}\ X\ \mathit{Term}_{\mathit{Pair}}\ 0_T\ (\beta,\gamma) = \beta$$
$$\textbf{type instance}\ X\ \mathit{Term}_{\mathit{Pair}}\ 1_T\ (\beta,\gamma) = \gamma$$

Again, the rules resemble selection functions, extracting the first and second components of the pair.

Summarising, quantified variables are represented as type families, and type equalities are encoded directly in the new C_{Eq} representation type. Type equalities on quantified variables need mobility rules, represented by type instances. We have seen this based on two example datatypes; in Section 6 we describe more formally how to encode indexed datatypes in the general case.

5. Instantiating generic functions

Now that we know how to represent indexed datatypes, we proceed to instantiate generic functions on these types. We split the discussion into generic consumers and producers, as they require a different approach.

5.1 Generic consumers

Instantiating generic equality to the *Vec* and *Term* types is unsurprising:

```
instance (GEq α) ⇒ GEq (Vec α ν) where
  geq = geqDefault
instance             GEq (Term α)  where
  geq = geqDefault
```

Using the instance for generic equality on C_{Eq} of Section 4.1.2, these instances compile and work fine. The instantiation of generic consumers to indexed datatypes is therefore no more complex than to standard datatypes.

5.2 Generic producers

Instantiating generic producers is more challenging, as we have seen in Section 3.2. For *Vec*, a first attempt could be:

```
instance (GEnum α) ⇒ GEnum (Vec α ν) where
  genum = genumDefault
```

However, this will always return the empty list: we do not know what ν is, so we cannot assume it to be *Ze*, *Su Ze*, or anything else. It could even be something nonsensical such as *Int*, so the only possible thing to return is the empty list. Instead, as before, we give two instances, one for *Vec* α *Ze*, and another for *Vec* α (*Su* ν), given an instance for *Vec* α ν:

```
instance (GEnum α) ⇒ GEnum (Vec α Ze) where
  genum = genumDefault
instance  ( GEnum α, GEnum (Vec α ν))
          ⇒ GEnum (Vec α (Su ν)) where
  genum = genumDefault
```

We can check that this works as expected by enumerating all the vectors of Booleans of length one: $genum :: [Vec\ Bool\ (Su\ Ze)]$ evaluates to $[Cons\ True\ Nil, Cons\ False\ Nil]$, the two possible combinations.

***Instantiating** $Term$.* Instantiating $GEnum$ for the $Term$ datatype follows a similar strategy. We must identify the types that $Term$ is indexed on. These are Int, $Bool$, and (α, β), in the Lit, $IsZero$, and $Pair$ constructors, respectively. The If constructor does not impose

any constraints on the index, and as such can be ignored for this purpose. Having identified the possible types for the index, we give an instance for each of these cases:

$$
\begin{array}{l}
\textbf{instance}\qquad GEnum\ (Term\ Int)\qquad \textbf{where}\\
\quad genum = genumDefault\\
\textbf{instance}\qquad GEnum\ (Term\ Bool)\quad \textbf{where}\\
\quad genum = genumDefault\\
\textbf{instance}\ (GEnum\ (Term\ \alpha), GEnum\ (Term\ \beta))\\
\qquad\qquad \Rightarrow GEnum\ (Term\ (\alpha, \beta))\ \textbf{where}\\
\quad genum = genumDefault
\end{array}
$$

We can now enumerate arbitrary $Term$s:

$$genum :: [Term\ Int]\ !!\ 5 \rightsquigarrow Pair\ (Lit\ 0)\ (IsZero\ (Lit\ 5))$$

However, having to write the three instances above manually is still a repetitive and error-prone task; while the method is trivial (simply calling $genumDefault$), the instance head and context still have to be given, but these are determined entirely by the shape of the datatype. We have written Template Haskell (Sheard and Peyton Jones 2002) code to automatically generate these instances for the user. In this section and the previous we have seen how to encode and instantiate generic functions for indexed datatypes. In the next section we look at how we automate this process, by analyzing representation and instantiation in the general case.

6. General representation and instantiation

In general, an indexed datatype has the following shape:

$$\begin{array}{l} \mathbf{data}\ D\ \overline{\alpha} = \forall\overline{\beta_1}.\overline{\gamma_1} \Rightarrow C_1\ \overline{\phi_1} \\ \qquad\qquad\qquad \vdots \\ \qquad\qquad\quad \mid\ \forall\overline{\beta_n}.\overline{\gamma_n} \Rightarrow C_n\ \overline{\phi_n} \end{array}$$

We consider a datatype D with arguments $\overline{\alpha}$ (which may or may not be indices), and n constructors $C_1 \ldots C_n$, with each C_i constructor potentially introducing existentially-quantified variables $\overline{\beta_i}$, type equalities $\overline{\gamma_i}$, and a list of arguments $\overline{\phi_i}$. We use an overline to denote sequences of elements.

We need to impose some further restrictions to the types we are able to handle:

1. Quantified variables are not allowed to appear as standalone arguments to the constructor: $\forall_{i,\beta\,\in\,\overline{\beta_i}}.\beta \notin \overline{\phi_i}$.

2. Indices are not allowed to appear as standalone arguments to a constructor: $\forall_{\alpha\,\in\,\overline{\alpha}}.\mathit{isIndex}\ \alpha\ D \rightarrow \forall_i.\alpha \notin \overline{\phi_i}$. We define *isIndex* in Section 6.2.

3. Quantified variables have to appear in the equality constraints:

$\forall_{i,\beta \in \overline{\beta_i}}.\exists\psi.\psi\ \beta \in \overline{\gamma_i}$ We require this to provide the mobility rules; in Section 7 we discuss how this restriction can be lifted.

For such a datatype, we need to generate two types of code:

1. The generic representation
2. The instances for generic instantiation

We deal with (1) in Section 6.1 and (2) in Section 6.2.

6.1 Generic representation

Most of the code for generating the representation is not specific to indexed datatypes; see, for instance, Magalhães et al. (2010) for a formalization of a similar representation. What needs to be adapted is the code generation for constructors, since now C_{Eq} takes two extra type arguments. The value generation (functions *from* and *to*) is not affected, only the representation type.

Type equalities. For each constructor C_i, an equality constraint $\tau_1 \sim \tau_2 \in \overline{\gamma_n}$ becomes the second and third arguments to C_{Eq}, for instance $C_{Eq} \ldots \tau_1\ \tau_2 \ldots$ Multiple constraints like $\tau_1 \sim \tau_2, \tau_3 \sim \tau_4$ become a product, as in $C_{Eq} \ldots (\tau_1 \times \tau_3)\ (\tau_2 \times \tau_4) \ldots$. An existentially-quantified variable β_i appearing on the right-hand side of a constraint of the form $\tau \sim \ldots$ or on the arguments to C_i is replaced by $X\ \gamma\ \iota\ \tau$, with γ the constructor representation type of C_i, and ι a type-level natural version of i.

Mobility rules. For the generated type families we provide the necessary mobility rules (Section 4.2.1). Given a constraint $\forall \overline{\beta_n}.\overline{\gamma_n}$, each equality $\beta \sim \psi\ \tau$, where $\beta \in \overline{\beta}$ and $\psi\ \tau$ is some type expression containing τ, we generate a **type instance** $X\ \gamma\ \iota\ (\psi\ \tau) = \tau$, where γ is the constructor representation type of the constructor where the constraint appears, and ι is a type-level natural encoding the index of the constraint. As an example, for the *Cons* constructor of Section 3.1, β is ν, τ is μ, ψ is *Su*, γ is Vec_{Cons}, and ι is 0_T.

6.2 Generic instantiation

To give a more thorough account of the algorithm for generation of instances, we will sketch its implementation in Haskell. We assume the following representation of datatypes:

```
data Datatype = Datatype [TyVar] [Con]
data Con      = Con Constraints [Type]

data Type     -- abstract
data TyVar    -- abstract

tyVarType :: TyVar → Type
```

A datatype has a list of type variables as arguments, and a list of constructors. Constructors consist of constraints and a list of arguments. For our purposes, the particular representation of types and type variables is not important, but we need a way to convert type variables into types (*tyVarType*).

Constraints are a list of (existentially-quantified) type variables and a list of type equalities:

```
data Constraints = [TyVar] ⋗ [TyEq]
data TyEq        = Type :∼: Type
```

We will need equality on type equalities, so we assume some standard equality on types and type equalities.

With this representation of datatypes we are ready to start the description of the algorithm for encoding indexed datatypes. We start by separating the datatype arguments into "normal" arguments and indices:

```
findIndices :: Datatype → ([TyVar + TyVar])
findIndices (Datatype vs cs) =
  [if v `inArgs` cs then L v else R v | v ← vs]
inArgs :: TyVar → [Con] → Bool
inArgs = ...
```

We leave *inArgs* abstract, but its definition is straightforward: it checks if the argument *TyVar* appears in any of the constructors as an argument. In this way, *findIndices* tags normal arguments with *L* and potential indices with *R*. These are potential indices because they could also just be phantom types, which are not only not used as argument but also have no equality constraints. In any case, it is safe to treat them as indices. The *isIndex* function used before is defined in terms of *findIndices*:

```
isIndex :: TyVar → Datatype → Bool
isIndex t d = R t ∈ findIndices d
```

Having identified the indices, we want to identify all the return types of the constructors, as these correspond to the heads of the instances we need to generate. This is the task of function *findRTs*:

```
findRTs :: [TyVar] → [Con] → [Constraints]
findRTs is []                         = []
findRTs is ((Con cts args) : cs) = let rs = findRTs is cs
                                     in if anyIn is cts
                                        then cts : rs
                                        else        rs

anyIn :: [TyVar] → Constraints → Bool
anyIn vs (_ ⇒ teqs) = or [v `in_TyEq` teqs | v ← vs]
in_TyEq :: TyVar → [TyEq] → Bool
in_TyEq = ...
```

We check the constraints in each constructor for the presence of a type equality of the form $i :\sim: t$, for some index type variable i and some type t. We rely on the fact that GADTs are converted to type equalities of this shape; otherwise we should look for the symmetric equality $t :\sim: i$ too.

Having collected the important constraints from the constructors, we want to merge those with the same return type. Given the presence of quantified variables, this is not a simple equality test;

we consider two constraints to be equal modulo all possible instantiations of the quantified variables:

```
instance Eq Constraints where
  (vs ⋗ cs) ≡ (ws ⋗ ds) = length vs ≡ length ws
                        ∧ cs ≡ subst ws vs ds
subst :: [TyVar] → [TyVar] → [TyEq] → [TyEq]
subst vs ws teqs = ...
```

Two constraints are equal if they abstract over the same number of variables and their type equalities are the same, when the quantified variables of one of the constraints are replaced by the quantified variables of the other constraint. This replacement is performed by *subst*; we do not show its code since it is trivial (given a suitable definition of *Type*).

Merging constraints relies on constraint equality. Each constraint is compared to every element in an already merged list of constraints, and merged if it is equal:

```
merge :: Constraints → [Constraints] → [Constraints]
merge c1              [] = [c1]
merge c1@(vs ⋗ cs) (c2@(ws ⋗ ds) : css)
  | c1 ≡ c2   = (vs ⋗ (cs ++ subst ws vs ds)) : css
  | otherwise = c2 : merge c1 css
mergeConstraints :: [Constraints] → [Constraints]
mergeConstraints = foldr merge []
```

We can now combine the functions above to collect all the merged constraints:

```
rightOnly :: [α + β] → [β]
rightOnly []         = []
rightOnly ((R a) : t) = a : rightOnly t
rightOnly (_     : t) =     rightOnly t

allConstraints :: Datatype → [Constraints]
allConstraints d@(Datatype _ cons) =
  let is = rightOnly (findIndices d)
  in mergeConstraints (findRTs is cons)
```

We know these constraints are of shape i :∼: t, where i is an index and t is some type. We need to generate instance heads of the form **instance** $G\ (D\ \overline{\alpha})$, where $\overline{\alpha} \in$ *buildInsts* D. The function *buildInsts* computes a list of type variable instantiations starting with the list of datatype arguments, and instantiating them as dictated by the collected constraints:

```
buildInsts :: Datatype → [[Type]]
buildInsts d@(Datatype ts _) = map (instVar ts) cs
```

```
    where cs = concat (map (λ(_ ⊳ t) → t) (allConstraints d))
instVar :: [TyVar] → TyEq → [Type]
instVar []       _     = []
instVar (v : vs) tEq@(i :∼: t)
    | tyVarType v ≡ i = t : map tyVarType vs
    | otherwise       = tyVarType v : instVar vs tEq
```

This completes our algorithm for generic instantiation to indexed datatypes. As mentioned before, the same analysis could be used to find out what *Read* instances are necessary for a given indexed datatype. Note however that this algorithm only finds the instance head for an instance; the method definition is trivial when it is a generic function.

7. Unrestricted indices

In the previous section we have seen how to handle datatypes with indices generically. We imposed three restrictions on the datatypes we handle. Restrictions (1) and (2) deal with existentially-quantified arguments; it is not the aim of this paper to add generic support for existentially-quantified datatypes. We discuss this issue in more detail in Section 9.1.

Restriction (3), on the other hand, seems a bit more arbitrary. A contrived example of a datatype which does not pass this restriction is the following:

```
data Tag α where
    Tag_I ::          Tag Int
    Tag_B :: Tag α → Tag Bool
```

Let us first write *Tag* with explicit quantification and constraints:

$$\begin{array}{lll}
\mathbf{data}\ \mathit{Tag}\ \alpha = & \alpha{\sim}\mathit{Int} & \Rightarrow \mathit{Tag}_I \\
\mid & \forall\beta.\alpha{\sim}\mathit{Bool} & \Rightarrow \mathit{Tag}_B\ (\mathit{Tag}\ \beta)
\end{array}$$

The constructor Tag_I simply sets the index α to *Int*. The constructor Tag_B sets the index to *Bool*, and also takes an argument tagged with any tag β. Valid values of type *Tag* are sequences of Tag_B's ending with a Tag_I.

Although this use of indices appears more simple than others we have shown before, the approach presented so far cannot handle these datatypes. Our first take would be to represent *Tag* as follows:

$$\begin{array}{l}
\mathbf{data}\ \mathit{Tag}_{\mathit{Tag}_I} \\
\mathbf{data}\ \mathit{Tag}_{\mathit{Tag}_B} \\
\mathbf{instance}\ \mathit{Constructor}\ \mathit{Tag}_{\mathit{Tag}_I}\ \mathbf{where}\ \mathit{conName}\ \_ = \texttt{"TagI"} \\
\mathbf{instance}\ \mathit{Constructor}\ \mathit{Tag}_{\mathit{Tag}_B}\ \mathbf{where}\ \mathit{conName}\ \_ = \texttt{"TagB"} \\
\\
\mathbf{instance}\ \mathit{Representable}\ (\mathit{Tag}\ \alpha)\ \mathbf{where} \\
\quad \mathbf{type}\ \mathit{Rep}\ (\mathit{Tag}\ \alpha) = \\
\qquad\quad C_{Eq}\ \mathit{Tag}_{\mathit{Tag}_I}\ \alpha\ \mathit{Int} \qquad U \\
\qquad + C_{Eq}\ \mathit{Tag}_{\mathit{Tag}_B}\ \alpha\ \mathit{Bool} \quad (\mathit{Rec}\ (\mathit{Tag}\ (X\ \mathit{Tag}_{\mathit{Tag}_B}\ 0_T\ \alpha))) \\
\quad \mathit{from}\ \mathit{Tag}_I \qquad\ = L\ (C_{Eq}\ U) \\
\quad \mathit{from}\ (\mathit{Tag}_B\ c) = R\ (C_{Eq}\ (\mathit{Rec}\ c)) \\
\quad \mathit{to}\ (L\ (C_{Eq}\ U)) \qquad\quad = \mathit{Tag}_I \\
\quad \mathit{to}\ (R\ (C_{Eq}\ (\mathit{Rec}\ c))) = \mathit{Tag}_B\ c
\end{array}$$

But this code does not type-check: the second equation of *from* introduces the constraint $\beta{\sim}X\ \mathit{Tag}_{\mathit{Tag}_B}\ 0_T\ \mathit{Bool}$ which we cannot prove. Since Tag_B does not put any constraints on β, we do not have any mobility rule, so we have no type instance for $X\ Tag_{Tag_B}\ 0_T\ \alpha$. The problem is that we have to come up with a particular instantiation for $X\ Tag_{Tag_B}\ 0_T\ \alpha$, while the constructor Tag_B accepts *any* index.

However, we can see from the declaration of Tag that the only possible instantiations for β are Int and $Bool$. In fact, we have already shown how to automatically determine the possible instan-

tiations of an index variable: this is the task of function *buildInsts* of Section 6.2. Therefore we argue the following equality should hold:

type instance $X\ Tag_{Tag_B}\ 0_T\ \alpha = Int + Bool$

However, this does not change the situation much: the second equation of *from* now requires the type equality $\beta \sim Int + Bool$, which we informally know is true, but cannot convince the type checker of.

7.1 Digression: proper kinds

In fact, all this trouble could be avoided if we had *user-defined kinds*. We are using type families and GADTs to perform type-level computations and make our programs very strongly typed. This means we make the structure of possible values well-defined and cleanly separated, but unfortunately the structure of the types themselves is not so organized: all types belong to a single kind $\star$.

This is clearly not what we want. Two examples follow:

- In Section 2.2, we say we define type classes to provide instances for the generic representation types. However, nothing prevents us from giving *GEq'* instances for any other types. Also, the compiler does not check that we have given instances for all the representation types; if we forget one instance, we only get an error (and a slightly obscure one) when instantiating the function to a particular type.
- When defining an indexed datatype, like *Vec*, we define its kind as being $\star \to \star \to \star$, while we hope the second parameter to only be instantiated with the types *Ze* or *Su*.

What we really need is the ability to create our own kinds and then restrict type arguments based on their kinds. Our representation types, for instance, should be grouped in a separate kind. We illustrate this with a hypothetical new language feature:

$$\begin{array}{l}
\textbf{kind}\ \star_R\ \textbf{where} \\
\quad \begin{array}{rl}
U & :: \star_R \\
+ & :: \star_R \to \star_R \to \star_R \\
\times & :: \star_R \to \star_R \to \star_R \\
C_{Eq} & :: \star_C \to \star \to \star \to \star_R \to \star_R \\
Var & :: \star \to \star_R \\
Rec & :: \star \to \star_R
\end{array}
\end{array}$$

We would also have a kind $\star_C$ containing only the generated datatypes for the constructor information. Furthermore, we can say that *Var* and *Rec* expect datatypes of kind $\star$ as their arguments.

A type class for defining a generic function would then explicitly state the kind of its argument:

$$\textbf{class}\ GEq'\ (\alpha :: \star_R)\ \textbf{where}\ \ldots$$

The type-checker could then check that all necessary instances were given.

Indexed datatypes could give proper kinds to their indices:

$$\begin{array}{l}
\textbf{kind}\ \star_T\ \textbf{where} \\
\quad \mathit{Int}\ \ :: \star_T \\
\quad \mathit{Bool} :: \star_T \\
\textbf{data}\ \mathit{Tag} :: \star_T \to \star\ \textbf{where} \\
\quad \mathit{Tag}_I :: \qquad\qquad\qquad\qquad\quad \mathit{Tag}\ \mathit{Int} \\
\quad \mathit{Tag}_B :: \forall \beta :: \star_T . \mathit{Tag}\ \beta \to \mathit{Tag}\ \mathit{Bool}
\end{array}$$

Note how Tag_B now constrains its argument to kind $\star_T$, effectively stating that it can only be *Int* or *Bool*. We would then hopefully be able to handle *Tag* generically by using *kind genericity* (albeit in a different style from Hinze (2002), as our kind structure is now richer).

However, while this style of programming is common practice in Ωmega (Sheard 2006), or in dependently-typed languages such as Agda (Norell 2007), it remains unclear how to integrate such features in Haskell, in particular regarding type and kind checking, inference, and interaction with all the other language features.

7.2 Unsafe coercions

Lacking better mechanisms for type-safe programming, we are left with telling the compiler it should just accept our code and stop complaining. In the second equation of *from* we have colored c in orange: this means that in the code we actually write *unsafeCoerce c*, where $\mathit{unsafeCoerce} :: \alpha \to \beta$ is a Haskell primitive to cast between values that are known to be equivalent.

Using *unsafeCoerce* means effectively abandoning the safety of Haskell; in theory we are now prone to runtime errors caused by using arguments of the wrong type. However, we are certain that our cast will not cause any problems because the variable is never used for purposes other than type-checking. In fact, as soon as we introduce the cast, the specific right-hand side of the **type instance** $X\ \mathit{Tag}_{\mathit{Tag}_B}\ 0_T\ \alpha$ is not important; the instance might even be absent altogether. The consequence is that the generic representation type $\mathit{Rep}\ (\mathit{Tag}\ \alpha)$ is not isomorphic to $\mathit{Tag}\ \alpha$; ignoring undefined values, $\mathit{Tag}\ \alpha$ might contain more inhabitants than $\mathit{Rep}\ (\mathit{Tag}\ \alpha)$, depending on the instance for $X\ \mathit{Tag}_{\mathit{Tag}_B}\ 0_T\ \alpha$.

Consider the possible instantiation **type instance** $X\ \mathit{Tag}_{\mathit{Tag}_B}\ 0_T\ \alpha = \mathit{Int}$. In this case, it is clear that $\mathit{Tag}\ \alpha$ contains more inhabitants than its representation, as the value $\mathit{Tag}_B\ (\mathit{Tag}_B\ \mathit{Tag}_I)$, for instance, cannot be represented. Its representation would be $R\ (C_{Eq}\ (\mathit{Rec}\ (\mathit{Tag}_B\ \mathit{Tag}_I)))$, but this does not type-check because we cannot prove that $\mathit{Tag}\ \mathit{Int} \sim \mathit{Tag}\ \mathit{Bool}$. However, we know we only have values of the generic representation types in two cases:

1. Converted with *from* from a user datatype value;
2. Produced from a generic producer such as *genum*.

Case (1) is handled by the unsafe cast: values of type *Tag* are, by definition, type-correct and can be converted to the generic representation because we cast them. Case (2) is more problematic: in the concrete case of *genum*, it means we might not be generating all possible values. This can happen because we first generate values of the generic representation type and then convert these with *to*. If the representation type is "too small", we might not generate all possible cases.

7.3 Generic instantiation

We can verify this easily by instantiating *GEnum* to *Tag*:

```
instance GEnum (Tag Int)    where
  genum = genumDefault
instance GEnum (Tag Bool) where
  genum = genumDefault
```

At this stage, the type-checker complains we lack an instance for $GEnum\ (Tag\ (X\ Tag_{Tag_B}\ 0_T\ \alpha))$. This is to be expected, as we use $Rec\ (Tag\ (X\ Tag_{Tag_B}\ 0_T\ \alpha))$ in the generic representation. We add such an instance:

```
instance GEnum (Tag (Int + Bool)) where
  genum = genumDefault
```

Note that we cannot use a type family in the instance head, so we replace $X\ Tag_{Tag_B}\ 0_T\ \alpha$ by its right-hand side $Int + Bool$.

This code type-checks correctly, but $genum :: [Tag\ Bool]$ returns $[\,]$. This should not be really surprising: the implementation of $genumDefault$ hits an instance $GEnum'\ (C_{Eq}\ Tag_{Tag_I}\ Int\ (Int + Bool)\ \alpha)$ and an instance $GEnum'\ (C_{Eq}\ Tag_{Tag_B}\ Bool\ (Int + Bool)\ \alpha)$; both return the empty list.

Draft version

What we want is to mimic the behavior of $GEnum'$ for sums in our $GEnum\ (Tag\ (Int + Bool))$ instance:

```
instance GEnum (Tag (Int + Bool)) where
  genum = gi ++ gb where
    gi  = genum :: [Tag Int]
    gb = genum :: [Tag Bool]
```

The enumeration now works as expected, with $take\ 2\ (genum :: [Tag\ Bool])$ returning $[Tag_B\ Tag_I, Tag_B\ (Tag_B\ Tag_I)]$. However, we are forced to use an unsafe cast one more time to convince the type checker to accept returning two lists of different types. We know this is “ok” because Int and $Bool$ are the two only possible instantiations of the index of Tag, so enumerating all possible terms for both indices and merging is the right implementation.

Lastly, it is worth mentioning that generic consumers such as GEq are not affected by our casts, and their instantiation remains trivial:

```
instance GEq (Tag α) where
  geq = geqDefault
```

7.4 Reflection

Circumventing the type system by using *unsafeCoerce* is not something to be taken lightly. Although we have explained informally why things cannot "go wrong" with our casts, we would still much prefer a well-kinded solution along the lines of Section 7.1. We do not consider the techiques described in this section to be part of the library, and we do not provide Template Haskell code to automatically introduce unsafe casts for the user. Instead, with this section we aim only at pointing out the feasability of generic programming for unrestricted indices, and providing a foundation for more structured future work.

8. Related work

Indexed datatypes can be seen as a subset of all GADTs, or as existentially-quantified datatypes using type-level equalities. Johann and Ghani (2008) developed categorical semantics of GADTs, including initial algebra semantics. While this allows for a better understanding of GADTs from a generic perspective, it does not translate directly to an intuitive and easy-to-use generic library.

Gibbons (2008) describes how to view abstract datatypes as existentially-quantified, and uses final coalgebra semantics to reason about such types. Rodriguez Yakushev and Jeuring (2010) describe an extension to the spine view (Hinze et al. 2006) supporting existential datatypes. Both approaches focus on existentially-quantified *data*, whereas we do not consider this case at all, instead focusing on (potentially existentially-quantified) *indices*. See Section 9.1 for a further discussion on this issue.

Within dependently-typed programming, indexing is an ordinary language feature which can be handled generically more easily due to the presence of type-level lambdas and explicit type application (e.g. Chapman et al. (2010); Morris (2007)).

9. Future work

In this section we discuss two possible extensions to the techniques described in the paper. Both extensions further increase the number of datatypes that can be handled generically using the library.

9.1 Existentials as data

While we can express indexed datatypes as GADTs or existentially-quantified datatypes with type-level equalities, the reverse is not true in general. Consider the type of dynamic values:

$$\textbf{data}\ \mathit{Dynamic} = \forall \alpha. \mathit{Typeable}\ \alpha \Rightarrow \mathit{Dyn}\ \alpha$$

The constructor *Dyn* stores a value on which we can use the operations of type class *Typeable*, which is all we know about this value. In particular, its type is not visible "outside", since *Dynamic* has no type variables. Another example is the following variation of *Term*:

```
data Term α where
  Const :: α                  → Term α
  Pair  :: Term α → Term β    → Term (α, β)
  Fst   :: Term (α, β)        → Term α
  Snd   :: Term (α, β)        → Term β
```

Here, the type argument α is not only an index but it is also used as data, since values of its type appear in the *Const* constructor. Our approach cannot currently deal with such datatypes. We plan to investigate if we can build upon the work of Rodriguez Yakushev and Jeuring (2010) to also support existentials when used as data.

9.2 Kind-generic programming with user-defined kinds

As discussed in Section 7.1, having user-defined kinds would allow for much safer type-level programming and more expressive programs. However, it remains to be seen how user-defined kinds can be introduced in Haskell, and how they can be used for generic programming. This is a promising direction for future research.

10. Conclusion

In this paper we have seen how to increase the expressiveness of a generic programming library by adding support for indexed datatypes. We have used the `instant-generics` library for demonstrative purposes, but we believe the technique readily generalizes to all other generic programming libraries using type-level generic representation and type classes. We have shown how indexing can be reduced to type-level equalities and existential quantification. The former is easily encoded in the generic representation, and the latter can be handled by encoding the restrictions on the quantified variables as relations to the datatype index. All together, our work brings the convenience and practicality of datatype-generic programming to the world of indexed datatypes, widely used in many applications but so far mostly ignored by boilerplate-removing strategies. We also hope to have illustrated the need and the potential advantages of a better kind system for Haskell.

Acknowledgments

This work has been partially funded by the Portuguese Foundation for Science and Technology (FCT) via the SFRH/BD/35999/2007 grant. We thank W. Bas de Haas for his helpful comments.

References

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

James Chapman, Pierre-Évariste Dagand, Conor McBride, and Peter Morris. The gentle art of levitation. In *ICFP'10*, pages 3–14. ACM, 2010.

Jeremy Gibbons. Datatype-generic programming. In Roland Backhouse, Jeremy Gibbons, Ralf Hinze, and Johan Jeuring, editors, *Datatype-Generic Programming*, volume 4719 of *LNCS*, pages 1–71. Springer, 2007.

Jeremy Gibbons. Unfolding abstract datatypes. In *MPC'08: Proceedings of the 9th international conference on Mathematics of Program Construc-*

tion, pages 110–133. Springer-Verlag, 2008.

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

Ralf Hinze, Andres Löh, and Bruno C. d. S. Oliveira. "Scrap Your Boilerplate" reloaded. In *FLOPS'06*, volume 3945 of *LNCS*. Springer, 2006.

Patricia Johann and Neil Ghani. Foundations for structured programming with GADTs. In *POPL'08: Proceedings of the 35th annual*

ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pages 297–308. ACM, 2008.

José Pedro Magalhães and W. Bas de Haas. Experience report: Functional modelling of musical harmony. Technical Report UU-CS-2011-007, Department of Information and Computing Sciences, Utrecht University, 2011.

José Pedro Magalhães, Atze Dijkstra, Johan Jeuring, and Andres Löh. A generic deriving mechanism for Haskell. In *Haskell '10: Proceedings of the third ACM Haskell symposium on Haskell*, pages 37–48, New York, NY, USA, 2010. ACM.

Peter Morris. *Constructing Universes for Generic Programming*. PhD thesis, University of Nottingham, November 2007.

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

Thomas van Noort, Alexey Rodriguez Yakushev, Stefan Holdermans, Johan Jeuring, Bastiaan Heeren, and José Pedro Magalhães. A lightweight approach to datatype-generic rewriting. *Journal of Functional Programming*, 20(Special Issue 3-4):375–413, 2010.

Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, September 2007.

Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. Simple unification-based type inference for GADTs. In *ICFP'06*, volume 41, pages 50–61. ACM, September 2006.

Alexey Rodriguez Yakushev and Johan Jeuring. Enumerating well-typed terms generically. In Ute Schmid, Emanuel Kitzelmann, and Rinus Plasmeijer, editors, *Approaches and Applications of Inductive Programming*, volume 5812 of *Lecture Notes in Computer Science*, pages 93–116. Springer, 2010.

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

Alexey Rodriguez Yakushev, Stefan Holdermans, Andres Löh, and Johan Jeuring. Generic programming with fixed points for mutually recursive datatypes. In *ICFP'09*, pages 233–244. ACM, 2009.

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

Tim Sheard. Generic programming programming in Ωmega. In Roland Backhouse, Jeremy Gibbons, Ralf Hinze, and Johan Jeuring, editors, *Datatype-Generic Programming*, volume 4719 of *Lecture Notes in Computer Science*, pages 258–284. Springer, 2006.

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