A Technical Overview of **Generic HASKELL**

Jan de Wit

January 16, 2002
Overview of talk

- Introduction
- Example
- Defining type-indexed values
- Kind-indexed types
- Type-indexed values
- The glue
- Conclusion
Introduction

- **Generic HASKELL** is a superset of Haskell designed for generic programming.
- Ideas pioneered by Ralf Hinze.
- First implementation as a source to source compiler by Jan de Wit and Andres Loeh.
Many programs work in essentially the same way, but on different data types. But still the code has to be written for each data type separately.

Examples:
- Summing or collecting all values in a data structure.
- Printing and parsing values.
- Systematically changing all values in a data structure (mapping).

The approach to generic programming we take works by *induction on the structure of types.*
Example: equality

- Checking whether two values of the same data type are equal is easy.
- Just use deriving `Eq` and `==`.
- That doesn’t always work. (higher-order datatypes)
Example: equality

► Checking whether two values of the same data type are equal is easy.

► Just use deriving `Eq` and `==`.

► That doesn’t always work. (higher-order datatypes)

► We can follow a cookbook recipe:
  - Check whether the two values are in the same alternative:
    - If not, they are not equal.
    - Otherwise, they are only equal if all components are equal.

► The last item needs to be checked by using appropriate equality functions.
Example: equality

For this data type:

```haskell
data Tree = Leaf Int
           | Node Tree Int Tree
```

the equality function looks like:

```haskell
eqTree :: Tree -> Tree -> Bool
eqTree (Leaf _ ) (Node _ _ _ ) = False
eqTree (Node _ _ _ ) (Leaf _ ) = False
eqTree (Leaf n1) (Leaf n2) = n1 == n2
eqTree (Node l1 n1 r1) (Node l2 n2 r2) = eqTree l1 l2 &&
                                           (n1 == n2) &&
                                           eqTree r1 r2
```
Example: equality

- If we abstract out the integers in the tree:

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

- Then we need to supply a function telling us how to compare values of type `a` (like the `..By` functions)

- The equality function now becomes:

```haskell
eqTree eqA (Leaf a1) (Leaf a2) = eqA a1 a2
eqTree eqA (Node l1 a1 r1) (Node l2 a2 r2) = eqTree eqA l1 l2 && eqA a1 a2 && eqTree eqA r1 r2
eqTree _ _ = False
```

- By modifying the recursive use of `eqA` we can deal with non-regular datatypes.
Defining type-indexed values

► How can we define equality in general?

► We need to know how to handle
  - Different alternatives: disjoint sums.
  - Components in a constructor: tuples.
  - Constructors and field labels.
  - Primitive types.
  - Function space constructors.

► We must be able to rewrite every data type using the above constructs.

► The transformations are straightforward to do by hand. If you want to have a compiler perform them in a structured way, on the other hand...
Defining type-indexed values

Disjoint sums:

```haskell
data Sum a b = Inl a | Inr b
```

Like the Either type.

The code:

```haskell
eq { | :+: | } :: (a -> a -> Bool) -> (b -> b -> Bool) ->
                  (Sum a b -> Sum a b -> Bool)

eq { | :+: | } eqA eqB (Inl a1) (Inl a2) = eqA a1 a2
eq { | :+: | } eqA eqB (Inl a1) (Inr b2) = False
eq { | :+: | } eqA eqB (Inr b1) (Inl a2) = False
eq { | :+: | } eqA eqB (Inr b1) (Inr b2) = eqB b1 b2
```
Defining type-indexed values

Products (tuples):

```
data Prod a b = a :*: b
```

Special case for 0-tuples:

```
data Unit = Unit
```

The code:

```
eq {| Unit |} Unit Unit = True
eq {| :*: |} eqA eqB (a1 :*: b1) (a2 :*: b2) =
    eqA a1 a2 && eqB b1 b2
```
Defining type-indexed values

Constructors:

```haskell
data Con a = Con ConDescr a
data ConDescr = ConDescr
  { conName :: String,
    ... }
```

ConDescr contains information about the constructor, such as:

- Its name,
- Its number of arguments,
- Whether it has field labels,
- ...
Defining type-indexed values

Constructors:

```haskell
data Con a = Con ConDescr a
data ConDescr = ConDescr
    { conName :: String
    , ...
    }
```

ConDescr contains information about the constructor, such as:
- Its name,
- Its number of arguments,
- Whether it has field labels,
- ...

Equality does not need this information:

```haskell
eq { | Con c |} eqA (Con _ a1) (Con _ a2) = eqA a1 a2
```

The same approach works for field labels.
Defining type-indexed values

► Primitive types:

```
eq {| Int |} = (==)
eq {| Char |} = (==)
eq {| IO |} = error "equality not defined for IO types!"
```
Defining type-indexed values

- Primitive types:

  
  \[
  \text{eq } \{ |\text{ Int } |\} = (==)
  \]
  \[
  \text{eq } \{ |\text{ Char } |\} = (==)
  \]
  \[
  \text{eq } \{ |\text{ IO } |\} = \text{error "equality not defined for IO types!"}
  \]

- You can override definitions:

  \[
  \text{data Set } a = \text{Set } [a]
  \]
  \[
  \text{setEq} :: (a \rightarrow a \rightarrow \text{Bool}) \rightarrow (\text{Set } a \rightarrow \text{Set } a \rightarrow \text{Bool})
  \]
  \[
  \text{eq } \{ |\text{ Set } |\} = \text{setEq}
  \]

- Needed here, otherwise Set equality would not have the expected properties.

- Overriding built-in types is also possible (for efficiency, perhaps):

  \[
  \text{eq } \{ |\text{ [] } |\} \text{ eqA } xs \text{ ys } = \ldots
  \]
Recall that

\[
data \text{ Tree } a = \text{ Leaf } a \\
| \text{ Node } (\text{ Tree } a) a (\text{ Tree } a)
\]

We now have everything we need to describe the top-level structure of this type.

\[
\text{type TreeStructure } a = \\
\text{ Sum (Con } a) \\
| \text{ (Con (Prod } (\text{ Tree } a) \\
| \text{ (Prod } a (\text{ Tree } a)))
\]

\[
\text{conLeaf} = \text{ ConDescr } "\text{Leaf}" \ldots \\
\text{conNode} = \text{ ConDescr } "\text{Node}" \ldots
\]

Note that this type is not recursive.
The equality function for TreeStructure is easy to write. It follows the definition of TreeStructure exactly:

```
eqTreeStructure eqA =
    eqSum (eqCon conLeaf eqA)
    (eqCon conNode (eqProd (eqTree eqA)
                    (eqProd eqA (eqTree eqA))))
```

Only the Con (and Label) cases have to be given some extra information.
Structure types in general

The datatype

```haskell
data T a1 ... an = C1 t1 ... tm 
| ... 
| Ck s1 ... sl
```

Has structure type

```haskell
type T__ a1 ... an = 
  Sum (Con (Prod t1 (Prod ... tm)))
  (Sum ...
   (Con (Prod s1 (Prod ... sl))))
```

A constructor without components corresponds to `Con Unit`

If a constructor has field labels every component gets wrapped in `Label`. 
If we suppose that

\[
eq \text{Tree} :: (a \to a \to \text{Bool}) \to \text{Tree } a \to \text{Tree } a \to \text{Bool}
\]

as required,
If we suppose that

\[
eq_{\text{Tree}} :: (a \to a \to \text{Bool}) \to \text{Tree} a \to \text{Tree} a \to \text{Bool}
\]
as required,

then we can infer that

\[
eq_{\text{TreeStructure}} :: (a \to a \to \text{Bool}) \to \\
\text{TreeStructure} a \to \\
\text{TreeStructure} a \to \\
\text{Bool}
\]

How can these two types be reconciled?
First of all, the data types Tree and TreeStructure are isomorphic via the pair of functions:

```haskell
fromTree :: Tree a -> TreeStructure a
fromTree (Leaf a) = Inl (Con conLeaf a)
fromTree (Node t1 a t2) = Inr (Con conNode (t1 :*: (a :*: t2)))
```

and

```haskell
toTree :: TreeStructure a -> Tree a
toTree (Inl (Con _ a)) = Leaf a
toTree (Inr (Con _ (t1 :*: (a :*: t2)))) = Node t1 a t2
```

These are trivial to generate.
A first sniff of glue

► We can use the isomorphism to make the type of \( eqTree \) match what we want:

\[
\text{eqTree } eqA \ t1 \ t2 = \text{eqTreeStructure } eqA \ (\text{fromTree } t1) \\
(\text{fromTree } t2)
\]

► This works for every combination of a data type and its structure type.

► For types other than \Tree, we might have to change the number of parameters.
We can use the isomorphism to make the type of `eqTree` match what we want:

\[
\text{eqTree } eqA \text{ t1 t2} = \text{eqTreeStructure } eqA \text{ (fromTree } \text{ t1)} \\
\text{ (fromTree } \text{ t2)}
\]

This works for every combination of a data type and its structure type.

For types other than `Tree`, we might have to change the number of parameters.

This scheme has to be modified if `Tree` appears on the right-hand side of the type, or inside a type constructor.

More on this later.
Kind-indexed types

The equality function has a type that varies along with the data type it operates on.

For types without type arguments:

\[
eq_T :: T \rightarrow T \rightarrow \text{Bool}
\]

For types with one simple type argument:

\[
eq_T :: (a \rightarrow a \rightarrow \text{Bool}) \rightarrow (T a \rightarrow T a \rightarrow \text{Bool})
\]

For types with two simple type arguments:

\[
eq_T :: (a \rightarrow a \rightarrow \text{Bool}) \rightarrow (b \rightarrow b \rightarrow \text{Bool}) \rightarrow (T a b \rightarrow T a b \rightarrow \text{Bool})
\]

And so on...
Kind-indexed types

► For higher-order types, the type of `eq` gets more complicated:

```haskell
data IntsChars ff = Ints (ff Int) | Chars (ff Char)
eqIntsChars eqFF (Ints ff1) (Ints ff2) = eqFF ff1 ff2
eqIntsChars eqFF (Chars ff1) (Chars ff2) = eqFF ff1 ff2
eqIntsChars eqFF _ _ = False
```

► To make this work we need the following type for `eqIntsChars`:

```haskell
(\forall a. \text{ff } a \rightarrow \text{ff } a \rightarrow \text{Bool}) \rightarrow
\text{IntsChars ff } \rightarrow \text{IntsChars ff } \rightarrow \text{Bool}
```

► We also need to tell the first equality function how to compare values of type `a`, so the final type becomes:

```haskell
(\forall a. (a \rightarrow a \rightarrow \text{Bool}) \rightarrow \text{ff } a \rightarrow \text{ff } a \rightarrow \text{Bool}) \rightarrow
\text{IntsChars ff } \rightarrow \text{IntsChars ff } \rightarrow \text{Bool}
```
**Kind-indexed types**

For types with one type constructor argument, i.e. kind \((* \rightarrow *) \rightarrow *\):

\[
eq T :: (\text{forall } a. (a \rightarrow a \rightarrow \text{Bool}) \rightarrow (\text{ff } a \rightarrow \text{ff } a \rightarrow \text{Bool})) \rightarrow (T \text{ ff } \rightarrow T \text{ ff } \rightarrow \text{Bool})
\]

The way to write the general pattern is:

\[
\text{type } \text{Eq } {{[ * ]}} t = t \rightarrow t \rightarrow \text{Bool}
\]

\[
\text{type } \text{Eq } {{[ k \rightarrow l ]}} t = \text{forall } u. \text{Eq } {{[ k ]}} u \rightarrow \text{Eq } {{[ l ]}} (t u)
\]

The type of \(\text{eq}\) becomes:

\[
\text{eq } {{| t :: k |}} :: \text{Eq } {{[ k ]}} t
\]
The general form of a kind-indexed type is:

type Poly {{ \ast }} t_1...t_n a_1...a_m = ...
type Poly {{ k \to l }} t_1...t_n a_1...a_m = \forall u_1...u_n .
  Poly {{ k }} u_1 ... u_n a_1...a_m ->
  Poly {{ l }} (t_1 u_1)...(t_n u_n) a_1...a_m

Poly has n varying type arguments, and m constant ones. In the equality example, n is 1 and m is 0.

Poly {{ k }} is a type constructor and its kind is:

Poly {{ k }} :: k -> k -> ... ->
  k_1 -> ... -> k_m ->
  \ast
Translating kind-indexed types

► This is a straightforward expanding process. The resulting type is simplified as much as possible.

► Remove unnecessary quantifications.

\[(\forall x. t) = t\]

if \(x\) doesn’t appear in \(t\).

► Move up quantifications on the right-hand side of a function arrow.

\[t \to (\forall x. s) = (\forall x. t \to s)\]

(avoid capture of \(x\) by renaming)

► Suppress quantifications at top level.

\[(\forall x. t) = t\]

► However in the presence of higher-order kinds these steps do not guarantee a type that is acceptable to Hugs or GHC.
In general, the definition of a type-indexed type looks like:

```haskell
poly {| t :: k |} :: Poly {[ k ]} t ...
poly {| Unit |} = ...
poly {| :+: |} polyA polyB = ...
poly {| :*: |} polyA polyB = ...
poly {| Con c |} polyA = ...
poly {| Int |} = ...
```

And optionally:

```haskell
poly {| Label l |} polyA = ...
poly {| (->) |} polyA polyB = ...
```
The following code is generated once per definition:

```haskell
polyUnit :: Poly {[ * ]} Unit ...
polyUnit = ...
polySum :: Poly {[ * -> * -> *]} Sum ...
polySum polyA polyB = ...
```

And for each data type \( T \) of kind \( k \) we specialise \( \text{poly} \):

```haskell
polyT :: Poly {[ k ]} T ...
polyT__ :: Poly {[ k ]} T__ ...
```

\( T__ \) is the structure type of \( T \).

Generating \( \text{polyT}__ \) is straightforward: just follow the structure of \( T \).
polyT is a wrapper around polyT__. The isomorphism between T and T__ can be extended to an isomorphism between Poly{[k]} T and Poly{[k]} T__.

How this is done depends in an essential way on the base case of the kind-indexed type Poly{[*]} T.

But fortunately, this can be handled by a generic function!

We combine the to and from functions forming the isomorphism into a single data type:

```haskell
data EP a b = EP { from :: a -> b, to :: b -> a }
```

So that we have for instance

```haskell
epTree :: EP (Tree a) (Tree__ a)
epTree = EP fromTree toTree
```
The generic function we will use to adapt the specialisation has type:

\[
\begin{align*}
\text{type Bimap } \{[ \ast ]\} & \quad \text{t1 t2} = \text{EP t1 t2} \\
\text{type Bimap } \{[ \, k \rightarrow l \, ]\} & \quad \text{t1 t2} = \forall u1 u2. \ \
\text{Bimap } \{[ \, k \, ]\} \ u1 \ u2 \ -> \ \text{Bimap } \{[ \, l \, ]\} \ (\text{t1 u1}) \ (\text{t2 u2}) \\
bimap \{| \ t :: k |\} :: \text{Bimap } \{[ \, k \, ]\} \ t \ t
\end{align*}
\]

So in particular:

\[
\begin{align*}
bimap \{| \ T :: \ast \rightarrow \ast |\} :: \text{EP a b} \rightarrow \text{EP (T a) (T b)} \\
bimap \{| \ T :: \ast \rightarrow \ast \rightarrow \ast |\} :: \text{EP a b} \rightarrow \text{EP c d} \rightarrow \text{EP (T a c) (T b d)}
\end{align*}
\]

Most cases in the definition of \(\text{bimap}\) are easy to write, they correspond to functor actions. Only the function case is interesting:

\[
\begin{align*}
bimap \{| \ (->) |\} \sim (\text{EP a2b b2a}) \sim (\text{EP c2d d2c}) = \text{EP} \\
{} \{ \text{from} = \ \\backslash a2c \rightarrow \ c2d . \ a2c . \ b2a \ , \ \text{to} = \ \\backslash b2d \rightarrow \ d2c . \ b2d . \ a2b \ \}
\end{align*}
\]
Suppose

type Poly \{[\,*\,]\} \ t = t \to \ Tree \ t

poly \{|\ t :: k \|\} :: Poly \{[\ k \}\} \ t

polyT :: T \to \ Tree \ T
polyT__ :: T__ \to \ Tree \ T__

The wrapper around polyT__ follows the type of the base case.

```
-- basecase t = Fun t (Tree t)
polyT = to (bimapFun epT (bimapTree epT)) (polyT__)
```
Overview of generated code

If we have the following **Generic HASKELL** source file:

```haskell
data T a b = ...

type Poly {[ * ]} t = ...

type Poly {[ k -> l ]} t = ...

poly { | t :: k | } :: Poly {[ k ]} t

poly { | Unit | } = ...

poly { | :+ : | } polyA polyB = ...
```
Overview of generated code

Then the following will be in the generated Haskell file:

```haskell
data T a b = ...
type T__ a b = ...

epT :: EP (T a b) (T__ a b)
bimapT :: EP a c -> EP b d -> EP (T a b) (T c d)
bimapT__ :: EP a c -> EP b d -> EP (T__ a b) (T__ c d)

polyUnit = ...
polySum polyA polyB = ...

polyT :: Poly {[ k ]} T
polyT a b = ... (polyT__ a b) ...
polyT__ :: Poly {[ k ]} T__
polyT__ a b = ... (polyT a b) ...
```

- `polyT__` only uses `polyT` if `T` is a recursive type.
Conclusion

- The basic idea behind generic programming is easy to grasp.
- **Generic HASKELL** provides a way to experiment with generic programming.
- The **Generic HASKELL** compiler takes care of the details which are
  - Not immediately apparent.
  - Potentially confusing.
  - Tedious and error-prone to code by hand.