A Technical Overview of **Generic H∀SKELL**

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 H∀SKELL** 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.

What is generic programming?

- 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 appropiate equality functions.

Example: equality

- For this data type:

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

- the equality function looks like:

```
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:

```
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:

```
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:

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

- Like the `Either` type.
- The code:

```
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:

```
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:

```
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:

```
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:

```
eq {| Int |} = (==)
eq {| Char |} = (==)
eq {| IO |} = error "equality not defined for IO types!"
```

- You can override definitions:

```
data Set a = Set [a]
setEq :: (a -> a -> Bool) -> (Set a -> Set a -> Bool)
eq {| Set |} = setEq
```

- Needed here, otherwise `Set` equality would not have the expected properties.
- Overriding built-in types is also possible (for efficiency, perhaps):

```
eq {| [] |} eqA xs ys = ...
```

Structure types

- Recall that

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

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

```
type TreeStructure a =
  Sum (Con a)
      (Con (Prod (Tree a)
                 (Prod a (Tree a))))
conLeaf = ConDescr "Leaf" ...
conNode = ConDescr "Node" ...
```

- Note that this type is not recursive.

Structure types

- 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

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

  Has structure type

```
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`.

Structure types

- If we suppose that

```
eqTree :: (a -> a -> Bool) -> Tree a -> Tree a -> Bool
```

as required,

Structure types

- If we suppose that

```
eqTree :: (a -> a -> Bool) -> Tree a -> Tree a -> Bool
```

  as required,

- then we can infer that

```
eqTreeStructure :: (a -> a -> Bool) ->
                   TreeStructure a ->
                   TreeStructure a ->
                   Bool
```

- How can these two types be reconciled?

A first sniff of glue

- First of all, the data types `Tree` and `TreeStructure` are isomorphic via the pair of functions

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

  and

```
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:

```
eqTree eqA t1 t2 = eqTreeStructure eqA (fromTree t1)
                                       (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.

A first sniff of glue

- We can use the isomorphism to make the type of `eqTree` match what we want:

```
eqTree eqA t1 t2 = eqTreeStructure eqA (fromTree t1)
                                       (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.
- 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:

```
eqT :: T -> T -> Bool
```

- For types with one simple type argument:

```
eqT :: (a -> a -> Bool) ->
       (T a -> T a -> Bool)
```

- For types with two simple type arguments:

```
eqT :: (a -> a -> Bool) ->
       (b -> b -> Bool) ->
       (T a b -> T a b -> Bool)
```

- And so on...

Kind-indexed types

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

```
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`:

```
(forall a. ff a -> ff a -> Bool) ->
  IntsChars ff -> IntsChars ff -> Bool
```

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

```
(forall a. (a -> a -> Bool) -> ff a -> ff a -> Bool) ->
  IntsChars ff -> IntsChars ff -> Bool
```

Kind-indexed types

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

```
eqT :: (forall a. (a -> a -> Bool) ->
                  (ff a -> ff a -> Bool)) ->
       (T ff -> T ff -> Bool)
```

- The way to write the general pattern is:

```
type Eq {[ * ]} t = t -> t -> Bool

type Eq {[ k -> l ]} t = forall u.
     Eq {[ k ]} u -> Eq {[ l ]} (t u)
```

The type of eq becomes:

```
eq {| t :: k |} :: Eq {[ k ]} t
```

Kind-indexed types

- The general form of a kind-indexed type is:

```
type Poly {[ * ]}      t1...tn a1...am = ...
type Poly {[ k -> l ]} t1...tn a1...am = forall u1...un .
  Poly {[ k ]}  u1     ... un      a1...am ->
  Poly {[ l ]} (t1 u1)...(tn un) a1...am
```

- 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 -> ... ->
                k1 -> ... -> km ->
                *
```

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 -> (forall x. s) = (forall x. t -> 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.

Type-indexed values

- In general, the definition of a type-indexed type looks like:

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

- And optionally:

```
poly {| Label l |} polyA = ...
poly {| (->) |} polyA polyB = ...
```

Type-indexed values

- The following code is generated *once* per definition:

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

- And for each data type `T` of kind `k` we specialise `poly`:

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

`T__` is the structure type of `T`.

- Generating `polyT__` is straightforward: just follow the structure of `T`.

Type-indexed values

- `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:

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

- So that we have for instance

```
epTree :: EP (Tree a) (Tree__ a)
epTree = EP fromTree toTree
```

Glue - bimaps

- The generic function we will use to adapt the specialisation has type:

```
type Bimap {[ * ]}       t1 t2 = EP t1 t2
type Bimap {[ k -> l ]} t1 t2 = forall u1 u2.
  Bimap {[ k ]} u1 u2 -> Bimap {[ l ]} (t1 u1) (t2 u2)
bimap {| t :: k |} :: Bimap {[ k ]} t t
```

- So in particular:

```
bimap {| T :: * -> * |} :: EP a b -> EP (T a) (T b)
bimap {| T :: * -> * -> * |} :: EP a b -> EP c d ->
                                EP (T a c) (T b d)
```

- Most cases in the definition of `bimap` are easy to write, they correspond to functor actions. Only the function case is interesting:

```
bimap {| (->) |} ~(EP a2b b2a) ~(EP c2d d2c) = EP
  { from = \a2c -> c2d . a2c . b2a
  , to   = \b2d -> d2c . b2d . a2b }
```

Glue - bimaps

- Suppose

```
type Poly {[ * ]} t = t -> Tree t

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

polyT :: T -> Tree T
polyT__ :: T__ -> 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 H∀SKELL** source file

```
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:

```
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 H∀SKELL** provides a way to experiment with generic programming.
- The **Generic H∀SKELL** compiler takes care of the details which are
  - Not immediately apparent.
  - Potentially confusing.
  - Tedious and error-prone to code by hand.