Generic Dictionaries

Ralf Hinze, Johan Jeuring

August ?, 2002

Why generic programming?

Why generic programming?

Because programs become easier to write!

Why generic programming?

Because programs become easier to write!

Why is it easier to write programs?

- Programs that you could only write in an untyped style have types now.
- Some programs are for free.
- Some programs are simple adjustments of library functions.

Introduction

Ralf has introduced the concept of a generic (or polytypic, type-indexed) function: a function that can be instantiated on all Haskell data types to obtain data specific functionality.

Over the last few years we have worked on generic programming and **Generic H∀SKELL**,see

```
http://www.generic-haskell.org/
```

The development of **Generic H∀SKELL** has to a large extent been example driven. In my lectures I will talk about three larger examples:

- Generic dictionaries
- XComprez, a compressor for XML documents
- The Zipper

Applications of generic programming

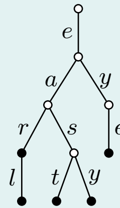
Applications of generic programming are found in several fields:

- Haskell's deriving construct: equality, read/show, ...
- Compiler tools: debuggers, garbage collectors, tracers.
- XML tools: editors, compressors, database tools.
- Typed term processing: rewriting, unification, pattern matching.
- Tree traversals.
- . . .

These fields have in common that structure drives the application. The programs are 'syntax-directed'.

Introduction Generic Dictionaries

A trie is a search tree scheme that employs the structure of a search key to organize information. For example, the set of strings $\{ear, earl, east, easy, eye\}$ can be represented by the following tree:



Overview

In this talk I will

- Define a module for generic dictionaries.
- Introduce type-indexed data types and kind-indexed kinds.
- Introduce default cases and constructor cases.
- Introduce dependencies and generic abstractions.

Dictionaries on strings

The data type of strings and dictionaries on strings can be defined as follows:

```
data String           =  Nil | Cons Char String

data FMap_String v  =
  Trie_String (Maybe v) (DictChar (FMap_String v))
```

where DictChar is a dictionary on characters. Here is the example:

```
eDict   :: FMap_String ()
eDict   =  Trie_String Nothing cDict1

cDict1  :: DictChar (FMap_String ())
cDict1  =  [('e',eDict1)]

eDict1  =  Trie_String Nothing cDict2
cDict2  =  [('a',eDict2),('y',eDict3)]
```

Looking up strings

Here is how you look up a string value.

```
lookup_String  ::  String -> FMap_String v -> Maybe v
lookup_String Nil        (Trie_String tn tc)  =  tn
lookup_String (Cons c s) (Trie_String tn tc)  =
  lookup_Char c tc  >>=  \t -> lookup_String s t

lookup_Char  ::  Char -> DictChar v -> Maybe v
```

Looking up characters

We implement a dictionary for characters as an ordered association list:

```
type DictChar v = [(Char,v)]

lookup_Char  ::  Char -> DictChar v -> Maybe v
lookup_Char c []                           =  Nothing
lookup_Char c ((c',v):xs) | c <  c'   =  Nothing
                          | c == c'   =  Just v
                          | c >  c'   =  lookup_Char c xs
```

Dictionaries on trees

A dictionary on trees is a value of the following data type.

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

data FMap_Tree fma v  =
  Trie_Tree (Maybe v) (FMap_Tree fma (fma (FMap_Tree fma v)))
```

The `lookup` function on tree dictionaries is defined as follows.

```
lookup_Tree  ::  (forall v.a -> fma v -> Maybe v) ->
                 Tree a -> FMap_Tree fma w -> Maybe w
lookup_Tree lua Leaf           (Trie_Tree tl tn)  =  tl
lookup_Tree lua (Node l m r) (Trie_Tree tl tn)  =
  lookup_Tree lua l tn  >>=  \tm ->
  lua m tm              >>=  \tr ->
  lookup_Tree lua r tr
```

A module for generic dictionaries

We want to have a function `FMap` that takes a data type and returns the dictionary type for that data type, together with functions for looking up in a dictionary, and manipulating a dictionary.

Furthermore, we want to have a number of functions on the generated dictionary data type

```
type FMap{| t |} ...

lookup{|t|}  ::  t -> FMap{| t |} v -> v
empty{|t|}   ::  FMap{| t |} v
single{|t|}  ::  (t,v) -> FMap{| t |} v
insert{|t|}  ::  (v -> v -> v) -> (t,v) ->
                 FMap{| t |} v -> FMap{| t |} v
delete{|t|}  ::  t -> FMap{| t |} v -> FMap{| t |} v
...
```

A type-indexed trie type

Tries are based on the following isomorphisms, also known as the laws of exponentials.

$$\begin{aligned} 1 \to_{\text{fin}} V &\cong V \\ (T_1 + T_2) \to_{\text{fin}} V &\cong (T_1 \to_{\text{fin}} V) \times (T_2 \to_{\text{fin}} V) \\ (T_1 \times T_2) \to_{\text{fin}} V &\cong T_1 \to_{\text{fin}} (T_2 \to_{\text{fin}} V) \end{aligned}$$

In **Generic H∀SKELL**, a type-indexed data type for dictionaries is defined as follows.

```
type FMap{| Unit |}         v  =  FMUnit (Maybe v)
type FMap{| Char |}         v  =  FMChar (DictChar v)
type FMap{| :+:  |} fma fmb v  =  FMEither (fma v,fmb v)
type FMap{| :*:  |} fma fmb v  =  FMProd (fma (fmb v))
```

Type-indexed data types

A type-indexed data type is a data type that is constructed in a generic way from an argument data type. Type-indexed data types are used in almost all applications given at the beginning of this lecture.

Besides finishing the example, I will also discuss the 'type' of type-indexed data types.

Type-indexed data types are just as important as type-indexed functions!

Kind-indexed kinds

What is the type of a type-indexed data type? The type of a data type is a kind. But a type-indexed data type depends on the kind of the input type. So a type-indexed data type has a kind-indexed kind!

```
type FMap{| t :: k |}  ::  FMAP{[ k ]}

FMAP{[  *   ]}  =  * -> *
FMAP{[ k->l ]}  =  FMAP{[ k ]} -> FMAP{[ l ]}
```

Function lookup

The generic function `lookup` is defined as follows:

```
lookup{| t :: k |}  ::  Lookup{[ k ]} t

type Lookup{[  *   ]} t v  =  t -> FMap{| t |} v -> Maybe v
type Lookup{[ k->l ]} t v  =
  forall a.Lookup{[ k ]} a v -> Lookup{[ l ]} (t a) v

lookup{| Unit |}            Unit     (FMUnit v)           =  v
lookup{| Char |}            c        (FMChar t)           =
  lookup_Char c t
lookup{| :+:  |} lua lub (Inl a) (FMEither (t1,t2))  =
  lua a t1
lookup{| :+:  |} lua lub (Inr b) (FMEither (t1,t2))  =
  lub b t2
lookup{| :*:  |} lua lub (a:*:b) (FMProd t)          =
  lua a t  >>=  \t' -> lub b t'
```

Default cases

Suppose I have a more efficient lookup function for characters `lookup_Char_eff`. Then I can reuse `lookup` by means of a *default case*:

```
lookup'{| t :: k |}    :: Lookup{[ k ]} t

lookup'{| Char |} c t  =  lookup_Char_efficient c t
lookup'{|   a  |}      =  lookup{| a |}
```

Constructor cases

Suppose I want to use the efficient lookup function only for one occurrence of the type `Char` in my data type. Then I can use a *constructor case*:

```
data Example = C1 Char | C2 Char | ....

lookup'{| t :: k |}                :: Lookup{[ k ]} t

lookup'{| case C1 |} (C1 c) t  =  lookup_Char_efficient c t
lookup'{|   a     |}           =  lookup{| a |}
```

Function empty

The generic function empty is defined as follows:

```
empty{| t :: k |}  ::  Empty{[ k ]} t

type Empty{[  *   ]} t v  =  FMap{| t |} v
type Empty{[ k->l ]} t v  =
  forall a.Empty{[ k ]} a v -> Empty{[ l ]} (t a) v

empty{| Unit |}            =  FMUnit Nothing
empty{| Char |}            =  FMChar []
empty{| :+:  |} ea eb      =  FMEither (ea,eb)
empty{| :*:  |} ea eb      =  FMProd ea
```

Function `single`: dependencies

The generic function `single` depends on the function `empty`:

```
dependency single <- single empty

type Single{[ * ]}     t  =  (t,v) -> FMap{| t |} v
type Single{[ k->l ]} t  =
  forall a.Single{[k]} a -> Empty{[k]} a -> Single{[l]} (t a)

single{| t :: k |}                          :: Single{[ k ]} t
single{| Unit |}               (Unit,v)      =  FMUnit (Just v)
single{| Char |}               (c,v)         =  FMChar [(c,v)]
single{| :+: |} sA eA sB eB (Inl a,v)      =
  FMEither (sA (a,v),eB)
single{| :+: |} sA eA sB eB (Inr b,v)      =
  FMEither (eA,sB (b,v))
single{| :*: |} sA eA sB eB (a :*: b,v)   =
  FMProd (sA (a,sB (b,v)))
```

Function `insert`: generic abstractions

The generic function `insert` can be defined as a generic abstraction of `single` and `merge`:

```
insert{| t :: * |}         :: (v -> v -> v) -> (t,v) ->
                                 FMap{| t |} v -> FMap{| t |} v
insert{| t |} c (x,v) d  =
  merge{| t |} c (single{| t |} (x,v)) d
```

The other functions on generic dictionaries are implemented in a similar fashion.

Exercise

The definition of function `insert` as a generic abstraction of `single` and `merge` limits its application to data types of kind $*$. Define `insert` as a type-indexed function with a kind-indexed type.

Conclusions

I have shown:

- How you can define a module for generic dictionaries.
- How you can define type-indexed data types in **Generic H∀SKELL**.
- How type-indexed data types are translated by **Generic H∀SKELL** to Haskell.

Next lectures

- XComprez
- The Zipper