Practical Generic Programming with OCaml

Jeremy Yallop

LFCS, University of Edinburgh

ML Workshop 2007
Instead of this ...

```ocaml

type α tree = Node of α | Branch of (α tree) × (α tree)

val show_tree : (α -> string) -> (α tree -> string)
let rec show_tree show_a = function
  Node a -> "Node " ^ show_a a
| Branch (l, r) -> "Branch ("^ show_tree l ^ ","
                    ^ show_tree r ^ ")"

show_list (show_pair (show_tree show_int) show_bool) t
```

You can write this!

```haskell
type α tree = Node of α | Branch of (α tree) × (α tree)
deriving (Show)
```

```haskell
Show.show<(int tree * bool) list> t
```
Outline

Basic idea

Customization

More customization: pickling

Conclusions
Haskell type classes as OCaml modules\textsuperscript{1}

```haskell
class Show a where
  show :: a → String

module type Show = sig
  type a
  val show : a → string
end
```

Type class as signature

\textsuperscript{1}Dreyer, Harper, Chakravarty and Keller. \textit{Modular Type Classes} (POPL 07)
Haskell type classes as OCaml modules

```ocaml
instance Show Int where
  show = showInt

module ShowInt :
  Show with type a = int =
struct
  type a = int
  let show = string_of_int
end
```

Instance as structure
Haskell type classes as OCaml modules

```ocaml
instance (Show a) => Show [a]
  where show l = "[" ++
       intersperse "," (map show l)
       ++ "]"

module ShowList (A : Show)
  : Show with type a = A.a list =
struct
  type a = A.a list
  let show l = "[
    concat "," (map A.show l)
    ^ "]"
end
```

Parameterized instance as functor
Haskell type classes as OCaml modules

```haskell
data Tree α = Node α | Branch (Tree α) (Tree α)
deriving (Show)

type α tree = Node of α | Branch of (α tree) × (α tree)
deriving (Show)
```
Haskell type classes as OCaml modules

\[
\text{data } \text{Tree } \alpha = \text{Node } \alpha \\
\quad | \text{Branch (Tree } \alpha) (\text{Tree } \alpha) \\
\text{deriving (Show)}
\]

\[
\text{type } \alpha \text{ tree} = \text{Node of } \alpha \\
\quad | \text{Branch of } (\alpha \text{ tree}) \times (\alpha \text{ tree}) \\
\text{deriving (Show)}
\]

\[
\Rightarrow
\]

\[
\text{instance Show } a \Rightarrow \text{Show (Tree } a) \\
\text{where} \\
\text{show} = \ldots
\]
Haskell type classes as OCaml modules

```
data Tree α = Node α
           | Branch (Tree α) (Tree α)
deriving (Show)
⇝
instance Show a => Show (Tree a)
  where
    show = ...
```

```
type α tree = Node of α
             | Branch of (α tree) × (α tree)
deriving (Show)
⇝
module Show_tree (A : Show)
  : Show with type a = A.a tree =
  struct
    type a = A.a tree
    let show = ...
  end
```
Haskell type classes as OCaml modules

```ocaml
show t
Show.show\tau{t}
```
Outline

Basic idea

Customization

More customization: pickling

Conclusions
Customization

type intset = int list
  deriving (Show)

Show.show<intset> [4; 1; 2; 3] ⇒ "[4; 1; 2; 3]"
**Customization**

```ocaml
type intset = int list

module Show_intset : Show.Show with type a = intset = Show.Defaults(struct
  let format fmt t =
    Format.fprintf fmt "{"%s}\n"
    (concat "," (map Show.show<int> (sort compare t)))
end)

Show.show<intset> [4; 1; 2; 3] ➞ "{1, 2, 3, 4}"```
Outline

Basic idea

Customization

More customization: pickling

Conclusions
More customization: pickling

- The “Pickle” class marshals values
- Pickle preserves and increases sharing wherever possible
- Sharing detection depends on the definition of equality
- We can customize pickling by customizing equality
What is equality?

- For values?
- For references?
- For functions?
- For user-defined types?
Sharing \( \lambda \) terms

type name = string
    deriving (Eq, Typeable, Pickle)

type exp = Var of name
    | App of exp \times exp
    | Lam of name \times exp
    deriving (Eq, Typeable, Pickle)
Sharing λ terms

type name = string
deriving (Typeable, Pickle)

type exp = Var of name
    | App of exp × exp
    | Lam of name × exp
deriving (Typeable, Pickle)

module Eq_name
  : Eq.Eq with type a = name =
    struct
      type a = name
      let eq = (=)
    end

module Eq_exp
  : Eq.Eq with type a = name =
    struct with type a = exp
      type a = exp
      let eq l r =
        (* α-equivalence *)
        ...
    end
Outline

Basic idea

Customization

More customization: pickling

Conclusions
Who can use *deriving*?

<table>
<thead>
<tr>
<th>Category</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Regular users</td>
<td><em>use</em> generic functions</td>
</tr>
<tr>
<td>Advanced users</td>
<td><em>customize</em> generic functions</td>
</tr>
<tr>
<td>Experts</td>
<td><em>write</em> generic functions</td>
</tr>
</tbody>
</table>
Coverage

Supported:
- base types
- variants
- tuples
- records
- mutable types
- polymorphic variants
- type aliases
- parameterized types
- (mutually) recursive types
- modules
- constraints (a bit)
- private types
- type replication

Not supported:
- non-regular recursion
- polymorphic record fields
- class types
- private rows
Remaining work

- more classes

- user-defined overloaded functions (not class methods)

```ocaml
(* print : Show \(\alpha\) =\(\rightarrow\) \(\alpha\) \rightarrow\) unit *)
let print\(<a:\text{Show}>\ v = \text{print\_endline}\ (\text{show}\<a>\ v)
```
Thank you!

http://code.google.com/p/deriving
(or google “ocaml” and “deriving”)