Code Reuse with Object-Encoded Transformers

Dmitry Boulytchev

Software Engineering Chair
Saint-Petersburg State University
dboulytchev@math.spbu.ru

TFP-2014
26-28 May 2014
Soesterberg, The Netherlands

ADT vs. Object Representation

ADT	Object Representation
Single type definition with multiple constructors	Multiple class definitions
Adding new transformation is easy	Adding new transformation is tedious
Adding new constructor is tedious	Adding new class is (rather) easy
Joining types is tough	Joining class hierarchies is tedious
Modifying existing transformation is tough	Modifying existing transformation is easy

Polymorphic Variants in OCaml

"The Expression Problem" [*Wadler, 1998*]

"Code Reuse Through Polymorphic Variants" [*Garrigue, 2002*]

```
type var = [`Var of string]

let evalVar s (`Var n) = s n

type 'a bin = [`Add of 'a * 'a | `Mul of 'a * 'a]

let evalBin eval = function
| `Add (x, y) -> eval x + eval y
| `Mul (x, y) -> eval x * eval y

type 'a expr = [var | 'a bin]

let evalExpr eval s = function
| #var as e -> evalVar s e
| #bin as e -> evalBin eval e

let rec eval s e = evalExpr (eval s) s e
```

Local Modification of Existing Transformation

Artificial example: counting number of leaves in a tree:

```
type tree = Leaf of string | Node of tree * tree

let rec leaves = function
| Leaf _ → 1
| Node (l, r) → leaves l + leaves r
```

Local Modification of Existing Transformation

Artificial example: counting number of leaves in a tree:

```
type tree = Leaf of string | Node of tree * tree

let rec leaves = function
| Leaf _ → 1
| Node (l, r) → leaves l + leaves r
```

Modification: do not count “empty leaves”:

Local Modification of Existing Transformation

Artificial example: counting number of leaves in a tree:

```
type tree = Leaf of string | Node of tree * tree

let rec leaves = function
| Leaf _ → 1
| Node (l, r) → leaves l + leaves r
```

Modification: do not count “empty leaves”:

```
let non_empty_leaves = function
| Leaf "" → 0
| t      → leaves t
```

Local Modification of Existing Transformation

Artificial example: counting number of leaves in a tree:

```
type tree = Leaf of string | Node of tree * tree

let rec leaves = function
| Leaf _ → 1
| Node (l, r) → leaves l + leaves r
```

Modification: do not count “empty leaves”:

```
let non_empty_leaves = function
| Leaf "" → 0
| t      → leaves t
```

In OO: visitors.

Object-Encoded Transformers

- Data representation is left untouched.
- Transformation is represented as object with method per each constructor.
- Traversal function to match against constructors and call appropriate methods.

Object-Encoded Transformers

Object-encoded transformer for counting leaves:

```
class leaves = object
  method c_Leaf l = 1
  method c_Node l r = l+r
end
```

Object-Encoded Transformers

Object-encoded transformer for counting leaves:

```
class leaves = object
  method c_Leaf l = 1
  method c_Node l r = l+r
end
```

Traversal function:

```
let rec transform t = function
| Leaf l → t#c_Leaf l
| Node (l, r) → t#c_Node (transform t l) (transform t r)
```

Object-Encoded Transformers

Object-encoded transformer for counting leaves:

```
class leaves = object
  method c_Leaf l = 1
  method c_Node l r = l+r
end
```

Traversal function:

```
let rec transform t = function
| Leaf l → t#c_Leaf l
| Node (l, r) → t#c_Node (transform t l) (transform t r)
```

Example:

```
let leaves = transform (new leaves)
let non_empty_leaves = transform
  object inherit leaves as super
    method c_Leaf l = if l = "" then 0 else super#c_Leaf l
  end
```

More Elaborated Version

- Transformation class: catamorphisms (folds/attribute grammar-defined transformations).
- One traversal function per type.
- Arguments of per-constructor methods are augmented with transformation functions.
- One abstract transformer (virtual class) to inherit from.
- Support for polymorphic ADT and polymorphic variant types.
- Syntax extension to generate all boilerplate code from type definitions.

Example: “show” and “eval”

```
@type expr = Var of string | Add of expr * expr | Const of int

class eval = object inherit [string -> int, int] @expr
  method c_Const _ _ n = n
  method c_Add s _ l r = l.fx s + r.fx s
  method c_Var s _ x   = s x
end

class show = object inherit [unit, string] @expr
  method c_Const _ _ n = string_of_int n
  method c_Add s _ l r = "Add (" ^ l.fx s ^ ", " ^ r.fx s ^ ")"
  method c_Var _ _ x   = "Var " ^ x
end

let eval = transform(expr) (new eval)
let show = transform(expr) (new show) ()
```

Example: Expression Problem

```
@type var = [`Var of string]
class ['v] var_eval = object inherit [string → 'v, 'v] @var
  method c_Var s _ x = s x
end

@type 'a bin = [ `Add of 'a * 'a | `Mul of 'a * 'a]
class ['a, 'b] bin_eval = object inherit ['a, int, 'b, int] @bin
  method c_Add s _ l r = l.fx s + r.fx s
  method c_Mul s _ l r = l.fx s * r.fx s
end

@type 'a expr = [ var | 'a arith ]
class ['a] expr_eval = object
  inherit ['a, int, string→int, int] @expr
  inherit [int] var_eval
  inherit ['a, string → int] bin_eval
end
```

Custom Traits/Plugins

- User-defined generators for concrete transformers.
- Dynamically loaded during syntax extension phase.
- Easy to implement for simple boilerplate transformations.
- Examples: `show`, `map`, `fold`.

Custom Traits/Plugins

- User-defined generators for concrete transformers.
- Dynamically loaded during syntax extension phase.
- Easy to implement for simple boilerplate transformations.
- Examples: `show`, `map`, `fold`.

```
@type tree = Leaf of string | Node of tree * tree with foldl

let leaves = transform(tree)
  object inherit [int] @fold[tree]
    method c_Leaf a _ _ = a+1
  end 0

let non_empty_leaves = transform(tree)
  object inherit [int] @fold[tree]
    method c_Leaf a _ l = if l = "" then a else a+1
  end 0
```

Custom Traits/Plugins

```
@type expr = Var   of string
           | Add   of expr * expr
           | Const of int with map

class simplify = object inherit @map[expr]
  method c_Add _ _ l r = match l.fx (), r.fx () with
  | Add (Const x, y), Add (Const z, t) → Add (Const (x+z), Add (y, t))
  | Add (Const x, y), Const z → Add (Const (x+z), y)
  | Const x, Add (Const y, z) → Add (Const (x+y), z)
  | Const x, Const y          → Const (x+y)
  | x      , (Const _ as y)   → Add (y, x)
  | x      , y                → Add (x, y)
end
```

Example: Lambda Reductions

"Demonstrating Lambda Calculus Reduction" [*Sestoft, 2002*]:

- categorization of steps;
- seven different reduction orders;
- big-step operational semantics;
- one-to-one correspondence between semantic specification and implementation code;
- seven similarly looking pieces of code (*actually, three*).

Example: Lambdas, Variables, Free Variables

```
@type lam =
| Var of string
| App of lam * lam
| Lam of string * lam with show, foldl
```

Example: Lambdas, Variables, Free Variables

```
@type lam =
| Var of string
| App of lam * lam
| Lam of string * lam with show, foldl

class var = object inherit [S.t] @foldl[lam]
  method c_Var s _ x = S.add x s
end

let vars = transform(lam) (new var) S.empty
```

Example: Lambdas, Variables, Free Variables

```
@type lam =
| Var of string
| App of lam * lam
| Lam of string * lam with show, foldl

class var = object inherit [S.t] @foldl[lam]
  method c_Var s _ x = S.add x s
end

let vars = transform(lam) (new var) S.empty

let fv = transform(lam)
  object inherit var
    method c_Lam s _ x l = S.union s (S.remove x ( l.fx S.empty ))
  end S.empty
```

Example: Object-Encoded Reducer

```
class virtual reducer = object inherit [unit, lam] @lam
  method virtual arg  : (unit, lam, lam, < >) a → lam
  method head : (unit, lam, lam, < >) a → lam = fun x → x.fx ()
  method c_Var _ x _ = ~:x
end
```

Example: Object-Encoded Reducer

```
class virtual reducer = object inherit [unit, lam] @lam
  method virtual arg  : (unit, lam, lam, < >) a → lam
  method head : (unit, lam, lam, < >) a → lam = fun x → x.fx ()
  method c_Var _ x _ = ~:x
end

let reduce r = transform(lam) r ()
```

Example: Dealing With Abstractions and Arguments

Example: Dealing With Abstractions and Arguments

```
class virtual reduce_under_abstractions =
  object inherit reducer
    method c_Lam _ _ x l = Lam (x, l.fx () )
  end

class virtual dont_reduce_under_abstractions =
  object inherit reducer
    method c_Lam _ s _ _ = ~:s
  end
```

Example: Dealing With Abstractions and Arguments

```
class virtual reduce_under_abstractions =
  object inherit reducer
    method c_Lam _ _ x l = Lam (x, l.fx () )
  end

class virtual dont_reduce_under_abstractions =
  object inherit reducer
    method c_Lam _ s _ _ = ~:s
  end

class virtual reduce_arguments =
  object inherit reducer
    method arg x = x.fx ()
  end

class virtual dont_reduce_arguments =
  object inherit reducer
    method arg x = ~:x
  end
```

Example: Strict vs. Non-strict

Example: Strict vs. Non-strict

```
class virtual strict g =
  object (this) inherit reducer
    method c_App _ s l m =
      match this#head l with
      | Lam (x, l') → s.f () (subst g x (m.fx ()) l')
      | l'          → let l'' = s.f () l' in
                      App (l'', this#arg m)
  end
```

Example: Strict vs. Non-strict

```
class virtual strict g =
  object (this) inherit reducer
    method c_App _ s l m =
      match this#head l with
      | Lam (x, l') → s.f () (subst g x (m.fx ()) l')
      | l'          → let l'' = s.f () l' in
                       App (l'', this#arg m)
  end

class virtual non_strict g =
  object (this) inherit reducer
    method c_App _ s l m =
      match this#head l with
      | Lam (x, l') → s.f () (subst g x ~:m l')
      | l'          → let l'' = s.f () l' in
                       App (l'', this#arg m)
  end
```

Example: Building Reduction Orders

Example: Building Reduction Orders

Call-by-name:

```
class bn g = object
  inherit dont_reduce_under_abstractions
  inherit dont_reduce_arguments
  inherit non_strict g
end

let bn l = reduce (new bn (context l)) l
```

Example: Building Reduction Orders

Call-by-name:

```
class bn g = object
  inherit dont_reduce_under_abstractions
  inherit dont_reduce_arguments
  inherit non_strict g
end

let bn l = reduce (new bn (context l)) l
```

Call-by-value:

```
class bv g = object
  inherit dont_reduce_under_abstractions
  inherit reduce_arguments
  inherit strict g
end

let bv l = reduce (new bv (context l)) l
```

Examlpe: Building Reduction Orders

Examlpe: Building Reduction Orders

Normal order:

```
class nor g = object
  inherit reduce_under_abstractions
  inherit reduce_arguments
  inherit non_strict g
  method  head x = bn ~:x
end

let nor l = reduce (new nor (context l)) l
```

Examlpe: Building Reduction Orders

Normal order:

```
class nor g = object
  inherit reduce_under_abstractions
  inherit reduce_arguments
  inherit non_strict g
  method  head x = bn ~:x
end

let nor l = reduce (new nor (context l)) l
```

Applicative order:

```
class ao c = object
  inherit bv c
  inherit reduce_under_abstractions
end

let ao l = reduce (new ao (context l)) l
```

Examlpe: Building Reduction Orders

Examlpe: Building Reduction Orders

Hybrid Applicative/Head Spine/Hybrid Normal Orders:

```
class ha c = object
  inherit ao c
  method head x = bv ~:x
end
let ha l = reduce (new ha (context l)) l

class he c = object
  inherit bn c
  inherit reduce_under_abstractions
end
let he l = reduce (new he (context l)) l

class hn c = object
  inherit nor c
  method  head x = he ~:x
end
let hn l = reduce (new hn (context l)) l
```

Conclusions

By now:

- lighweight datatype-generic framework;
- one per-type traversal function, one abstract object-encoded transformer (generated automatically);
- a number of plugins to generate concrete transformers for some widespread transformations (`show`, `map`, `fold`, `eq`, `compare`,...);
- an ability to modify existing transformations;
- an ability to “join” transformations for “joined” types.

Future work:

- find more applications;
- evaluate and address performance issues;
- implement more plugins (with contexts?).

Thank you!

- Source code:
  `https://code.google.com/p/generic-transformers`