Generic Programming with Combinators and Objects

Dmitrii Kosarev

St. Petersburg State University
JetBrains Research

Mini Conference 2018

Motivation

A dependency of *OCanren* which was a topic of last year

Main idea: extend the technique called *generic programming* with generation of *extensible* code

Has some attention in community: see François Pottier paper from ICFP2017

Our library should be better then others from engineering point view

Was presented on ML Workshop (colocated with ICFP 2018)

What and Why

Datatype-generic programming [Gibbons, 2006] — use types to define transformations

Two main approaches:

- generate a transformation from type definition (`deriving`, etc.)
- define a representation for all types, implement a transformation of interest in terms of representation, translate a concrete type into its representation [W.Swierstra, 2008], [Chakravarty *et al*, 2009]

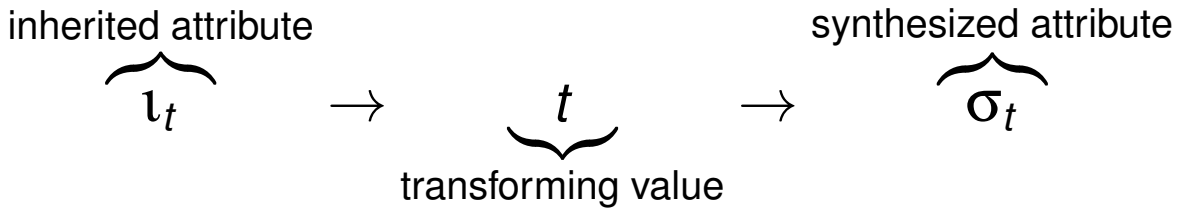
Observation: in many cases, a transformation of interest is a (small) modification of some other transformation

Approach: utilize object-oriented features to inherit one transformations from others

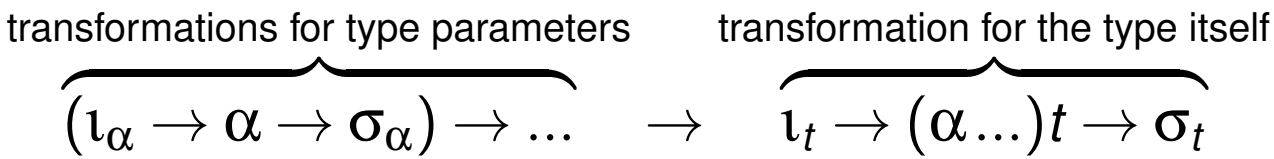
Attribute Grammars and Catamorphisms

Attribute grammars (AG) [Knuth, 1968] as a way of describing catamorphisms [Fokkinga *et al*, 1990], [Swierstra *et al*, 2000, 2009]

A common shape of all transformations for a type t:



For a parametric type $(\alpha \ldots)\ t$:



Transformation Function

A type:

```
type (α...) t = C of q * ... | ...
```

A *transformation function*:

```
let transform_t ω ι t =
  match t with
  | C (x, ...) → ω#C ι x ...
  ...
```

A *transformation object* ω: implements a transformation on a method-per-constructor basis

Transformation Class

An abstract class to mass-produce all transformation objects:

all type parameters for a transformation

$$\overbrace{\iota_\alpha, \alpha, \sigma_\alpha, \ldots, \iota, \sigma}$$

```
class virtual [ ... ] t_t =
  object
    method virtual C : ...
    ...
  end
```

An Observable Example: Generic Definitions

```
type expr = Const of int | Add of expr * expr
```

```
let transform_expr ω ι = function
| Const i      → ω#c_Const ι i
| Add  (x, y)  → ω#c_Add  ι x y
```

```
class virtual [ι, σ] expr_t =
object
  method virtual c_Const : ι → int  → σ
  method virtual c_Add   : ι → expr → expr → σ
end
```

An Observable Example: Concrete Transformation

```
class expr_show (fself : unit → expr → string) =
object
  inherit [unit, string] expr_t
  method c_Const () i =
    sprintf "Const (%d)" i
  method c_Add () x y =
    sprintf "Add (%s, %s)" (fself x) (fself y)
end

let show_expr e =
  fix (fun f → transform_expr (new expr_show f)) () e
```

Design Choices and Properties

The transformation function is shallow (non-recursive)

The transformation function is *not* parameterized by transformations for type parameters

A single per-type transformation function is used to implement all transformations for the type

All transformation objects can be inherited from a single class

For recursive types the knot has to be tied at the transformation object creation time

Subtleties

The types of transformation objects can be (and, as a rule, are) scary

The support for polymorphic variants requires extra work

The support for mutually-recursive types requires extra extra work

The `fix` can be improved to not to create an identical object for each node of the data structure to transform

Implementation Overview

The transformation function and abstract transformation class are generated by the framework from a type declaration

Supported: regular ADTs, type applications, structures, polymorphic variants

Not supported: extensible types (“ ... ”, “+=”), GADTs, module and object types

Plugins: predefined (`show`, `map`, `fold`, etc.) and user-defined

Available by means of both `camlp5` and `ppxlib`

Example: gmap

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

The default transformation `@expr[gmap]` just makes a copy

This default behavior can be overriden, obtaining a new transformation:

```
let substitute st e =
  fix (fun f →
        transform(expr) (object inherit [_] @expr[gmap] f
                           method c_Var _ x = Const (st x)
                         end)
                        ()
      )
  e
```

Example: `foldr`

```
@type expr =
| Var   of string
| Binop of expr * expr with foldr
```

The default transformation makes nothing (just threads the accumulator value in a bottom-up manner)

Again, by overriding only the “interesting case” a new transformation can be constructed:

```
module S = Set.Make (String)

let vars e = fix (fun f →
  transform(expr)
    (object inherit [S.t, expr] @expr[foldr] f
        method c_Var vs s = S.add s vs
      end)
  ) S.empty e
```

Example: Polymorphic Variant Support

We can equally use a polymorphic variant version (note, here we define a separate class rather than immediate object):

```
@type 'e base = [`Var of string | `Binop of 'e * 'e] with foldr

class ['e, 's] vars_base fself fe =
  object inherit ['e, S.t, 's] @base[foldr] fself fe
    method c_Var vs s = S.add s vs
  end
```

Example: Polymorphic Variant Support, More

We can declare another polymorphic variant type, now representing a binding construct, and implement a custom class:

```
@type 'e lam = [`Lam of string * 'e] with foldr

class ['e, 's] vars_lam fself fe =
  object inherit ['e, S.t, 's] @lam[foldr] fself fe
    method c_Lam vs s e = S.union vs (S.remove s @@ fe S.empty e)
  end
```

Example: Polymorphic Variant Support, Even More

... and combine these two types and those two classes:

```
@type 'e expr = ['e base | 'e lam] with foldr

class ['e, 's] vars_expr fself fe =
  object
    inherit ['e, 's] vars_base fself fe
    inherit ['e, 's] vars_lam  fself fe
  end

let vars e =
  fix (fun f → transform(expr) (new vars_expr f f)) S.empty e
```

P.S. Alternatives

Our work can be compared to Visitors from François Pottier. Two main differencies are:

- Universal quantification of methods in visitors
- A more explicit way to specify type parameters of a class in our approach

Universal quantification in methods leads to errors

```
(* no error yet *)
type ('a, 'b) t = Foo of 'a * 'b
[@@deriving visitors { variety = "map"; polymorphic = true }]
```

Universal quantification in methods leads to errors

```
(* no error yet *)
type ('a, 'b) t = Foo of 'a * 'b
[@@deriving visitors { variety = "map"; polymorphic = true }]

(* Code generation leads to compilation error below *)
type 'a t2 = ('a, int) t
[@@deriving visitors { variety = "map"; polymorphic = true
                     ; name="map1"}]

(* And here too *)
type 'a t2 = T2 of ('a, int) t [@@unboxed]
[@@deriving visitors { variety = "map"; polymorphic = true
                     ; name="map2" }]
```

Universal quantification in methods leads to errors

```
(* no error yet *)
type ('a, 'b) t = Foo of 'a * 'b
[@@deriving visitors { variety = "map"; polymorphic = true }]

(* Code generation leads to compilation error below *)
type 'a t2 = ('a, int) t
[@@deriving visitors { variety = "map"; polymorphic = true
                     ; name="map1"}]

(* And here too *)
type 'a t2 = T2 of ('a, int) t [@@unboxed]
[@@deriving visitors { variety = "map"; polymorphic = true
                     ; name="map2" }]
```

But in *monomorphic mode* visitors library works fine

```
type 'a t2 = T2 of ('a, int) t [@@unboxed]
[@@deriving visitors { variety = "map"; polymorphic = false
                     ; name="map3" }]
```

Type parameters of transformation classes

type $(\alpha ...)$ t = ...

In visitors:

```
class ['self] c = object (self: 'self) ... end
```

- In our library we have many type parameters for a class but there is an important one: the *opened* version of type t
  $\Rightarrow$ joining of polymorphic variants becomes doable.
- Visitors are failing to generate a signature.

Concluding Remarks

The framework is still in development, but the fixpoint is close

Performance issues: method invocation is slower, than a direct function call

Only homogeneous plugins are supported so far

Ad hoc polymorphism? Existentials?

Object/classes can be good, when cooked properly

Thank you!

Example: Hyperlinked OCaml Typedtree (18+)

```
type pattern = [%import: Typedtree.pattern]
and pat_extra = [%import: Typedtree.pat_extra]
and pattern_desc = [%import: Typedtree.pattern_desc]
and expression = [%import: Typedtree.expression]
and expression_desc = [%import: Typedtree.expression_desc]
and exp_extra = [%import: Typedtree.exp_extra]
...
```

(**53 more** mutually recursive types from the module `Typedtree`)

```
...
and 'a class_infos = [%import: 'a Typedtree.class_infos]
[@@deriving gt ~options: {html; fmt}]
```

Example: Patterns (Hyperlink Destinations)

```
class ['self] pattern_desc_link mut_trfs fself =
object
  inherit ['self] html_pattern_desc_t_stub mut_trfs fself
  method! c_Tpat_var () { Ident.name } nameloc =
    let loc_str =
      Location.show_location nameloc.Asttypes.loc
    in
    HTML.ul @@ HTML.seq
        [ HTML.anchor loc_str @@ HTML.string @@
          sprintf "%S from %s" name loc_str
        ]
end
```

Example: Identifiers (Hyperrefs)

```
class ['self] expression_with_link mut_trfs fself =
object
  inherit ['self] html_expression_t_stub mut_trfs fself as super

  method! do_expression () e =
    match e.exp_desc with
    | Texp_ident (p,lloc,vd) →
      let where = Location.show_t
        (Ocaml_common.Env.find_value p e.exp_env).val_loc
      in
      HTML.ref ("#" ^ where) @@ HTML.string @@
      sprintf "%s  from %s" (Longident.show_t lloc.txt) where
    | _ → super#do_expression () e
end
```

Example: Tying the Knot

```
let html_structure =
  let { html_structure } = html_fix_case
    ~expression0:
      { html_expression_func = new expression_with_link }
    ~pattern_desc0:
      { html_pattern_desc_func = new pattern_desc_with_link }
    ()
  in
  html_structure.html_structure_trf
```