OCanren Documentation

JetBrains Research

May 27, 2023

Contents

This library implements a framework for datatype-generic programming in Objective Caml language.

The key feature of the approach in question is object-oriented representation of transformations performed over regular algebraic datatypes. Our implementation supports polymorphic variants; in particular, a transformation for a “joined” polymorphic variant type can be acquired via inheritance from the transformations for its counterparts.

CHAPTER 1

See also

visitors

BAP's vistors

Janestreet's PPX Traverse

1.1 Usage

CHAPTER 2

As PPX

Use findlib package `GT.ppx` in combination with `ppxlib`. See `ppxlib`'s manual for full guidance. In short do

```
~ ocaml
        OCaml version 4.07.1

# #require "GT.ppx";;
../GT: added to search path
../GT/./pp_gt.native --as-ppx: activated
# type 'a list = Nil | Cons of 'a * 'a list [@@deriving gt ~options:{show}];;
```

CHAPTER 3

As Camlp5 syntax extension

Use findlib package `GT.syntax.all` to enable extension and all built-in plugins. To compile and see the generated code use the following command:

```
ocamlfind opt -syntax camlp5o -package GT.syntax.all regression/test081llist.ml -
↪dsource
```

To preprocess only the code in this library (for example, a test) use the following shell command:

```
(cd _build && ../camlp5o_pp.sh pr_o.cmo ../regression/test005.ml)
```

To use camlp5 (>= 7.12) syntax extension in toplevel try this:

```
#use "topfind.camlp5"
#camlp5o;;
#require "GT,GT.syntax.all";;
@type t = GT.int with gmap,show;; (* for example *)
```

3.1 Directory structure

- The framework for generation is in `common/`. The generic plugin for adding new transformations is in `common/plugin.ml`.
- All built-in plugins live in `plugins/` and depend on the stuff in `common/`.
- Camlp5-specific preprocessing plugin lives in `camlp5/`. Depend on stuff in `common/`.
- PPX-specific preprocessing plugin lives in `ppx/`. Depends on stuff in `common/`.
- Built-in plugins that represent transformations live in `plugins/`. Depends on `common/`.
- A library for built-in types and transformations for types from Pervasives live in `src/`. Depends on syntax extension from `camlp5/` and plugins from `plugins/`.

CHAPTER 4

Dependencies

- `ppxlib`
- `camlp5`
- `ocamlgraph` for topological sorting
- `dune` as build system

CHAPTER 5

Compilation

- `make` to compile whole library.
- `make && make tests` to compile regression tests too.

CHAPTER 6

Example: Processing Expressions

Let us have the following type for simple arithmetic expressions:

```
type expr =
  Add of expr * expr
| Mul of expr * expr
| Int of int
| Var of string
```

One of the first typical "boilerplate" tasks is printing; much like other available generic frameworks this simple goal can be achieved with our library by a little decoration of the original declaration:

```
type expr =
| Add of expr * expr
| Mul of expr * expr
| Int of GT.int
| Var of GT.string [@@deriving gt ~options:{show}]
```

For mutually recursive type declarations add decoration only to the last type

```
type t = ...
and heap = t [@@deriving gt ~options:{ show }]
```

We replaced here `int` and `string` with `GT.int` and `GT.string` respectively, and added `[@@deriving gt ~options:{show}]` to the end of type declaration to make the framework generate all "boilerplate" code for us. `GT.int` and `GT.string` are two synonyms for regular standard types, equipped with some additional generic features; alternatively, we could just add `open GT` to the beginning of the code snippet and use short names. Further we will continue to explicitly mention features of the framework in a fully-qualified form.

Having made this, we can instantly print expressions with the following (a bit cryptic) construct:

```
GT.transform(expr) (new show_expr_t) () (Mul (Var "a", Add (Int 1, Var "b")))
```

Here

- `GT.transform(expr)` - type-indexed function, applied to the type **expr**; in our framework all computations are performed by this single function;
- `new show_expr_t` - an expression, which creates a *transformation object*, encapsulating the "show" functionality for type `expr`;
- we provide unit value as additional parameter, which in fact is not used; think of it as an initial value for fold-like transformations;
- the rest is the expression tree we're going to show.

The result of this expression evaluation, as expected, is

```
Mul (Var (a), Add (Int (1), Var (b)))
```

In our framework (at least by now) all transformations are expressed by the following common pattern:

```
GT.transform(t) tr_obj init value_
```

or more precisely

```
GT.fix (fun fself init value_ ->
  GT.transform tree (new tr_class f_1 ... f_n fself) init value
) init value_
```

where

- *t* is a polymorphic type with *n* parameters;
- *tr_obj* – transformation object for some transformation;
- *f_1*, . . . , *f_n* – transformation functions for type parameters;
- *init* – an initial value for transformation (initial inherited attribute);
- *value* – the value to transform of type *(a_1, a_2, . . . , a_n) t*.

Transformations function *f_j* usually have type *inh_j -> a_j -> syn_j*. Types *inh_j* and *syn_j* may be arbitrary; they can be interpreted as **inherited** and **synthesized** attributes for type parameter transformations, if we interpret catamorphisms in attribute-grammar fashion. For example, for "show" *inh_j* is *unit* and *s_j* is *string*.

Transformation object is an object which performs the actual transformation on a per-constructor basis; we can think of it as a collection of methods, one per data type constructor. Transformation objects can be given either implicitly by object expressions or created as instances of transformation classes. Each class, in turn, can be generated by a system, hand-written from scratch or inherited from an existing ones.

In our example the phrase "*with show*" makes the framework to invoke a used-defined plugin, called "*show*". The architecture of the framework is developed to encourage the end-users to provide their own plugins; writing plugins is considered as an easy task.

The key feature of the approach we advocate here is that object-oriented representation of transformations makes them quite easy to modify. For example, if we are not satisfied by the "default" behavior of "show", we can adjust it only for the "cases of interest":

```
class show1 fself = object
  inherit show_expr_t fself
  method c_Var _ _ s = s
end

GT.fix (fun fself ->
  GT.transform tree (new show1 fself) ()
```

(continues on next page)

(continued from previous page)

```
)
(Mul (Var "a", Add (Int 1, Var "b")))
```

Now the result is

```
Mul (a, Add (Int (1), b))
```

We fixed only the "case of interest"; method `c_Var` takes three arguments - the inherited attribute (which is always unit here), the original value (actually, *augmented* original value, see below), and immediate arguments of corresponding constructor (actually, their *augmented* versions). In this case `s` is just a string argument of the constructor `Var`.

If we still not satisfied with the result, we can further proceed with fixing things up:

```
class show2 fself = object
  inherit show' fself
  method! c_Int () _ i = string_of_int i
end

GT.transform(expr) (new show2) () (Mul (Var "a", Add (Int 1, Var "b")))
```

The result now is

```
Mul (a, Add (1, b))
```

In the next step we are going to switch to infix representation of operators; this case is interesting since we have to adjust the behavior of the transformation not only for the single node, but to all its sub-trees as well. Fortunately, this is easy:

```
class show3 fself = object
  inherit show'' fself
  method c_Add _ _ x y = x.GT.fx () ^ " + " ^ y.GT.fx ()
  method c_Mul _ _ x y = x.GT.fx () ^ " * " ^ y.GT.fx ()
end

GT.transform(expr) (new show3) () (Mul (Var "a", Add (Int 1, Var "b")))
```

Method `c_Add` takes four arguments:

- inherited attribute (here it is `unit`);
- original node;
- parameters of the constructor (`x` and `y`).

Finally, we may want to provide a complete infix representation (including a minimal amount of necessary brackets):

```
class show4 fself =
  let enclose op p x y =
    let prio = function
      | Add (_, _) -> 1
      | Mul (_, _) -> 2
      | _ -> 3
    in
    let bracket f x = if f then "(" ^ x ^ ")" else x in
    bracket (p >  prio x) (fself () x) ^ op ^
    bracket (p >= prio y) (fself () y)
  in
  object
```

(continues on next page)

(continued from previous page)

```
    inherit show3 fself
    method c_Mul _ _ x y = enclose "*" 2 x y
    method c_Add _ _ x y = enclose "+" 1 x y
  end
```

On the final note for this example we point out that all these flavors of `show` transformation coexist simultaneously; any of them can be used as a starting point for further adjustments.

Our next example is variable-collecting function. For this purpose we add `foldl` the the list of user-defined plugins:

```
type expr =
| Add of expr * expr
| Mul of expr * expr
| Int of GT.int
| Var of GT.string
[@@deriving gt ~options:{ show; foldl } ]
```

With this plugin enabled we can easily express what we want:

```
module S = Set.Make (String)
class vars fself = object
  inherit [S.t] foldl_expr_t fself
  method c_Var s _ x = S.add x s
end

let vars e = S.elements (GT.transform(expr) (new vars) S.empty e
```

In the default version, `@expr[foldl]` is generated in such a way that inherited attribute value (in out case of type `S.t`) is simply threaded through all nodes of the data structure. This behavior as such gives us nothing; however we can redefine the "interesting case" (variable occurrence) to take this occurrence into account.

The next example - expression evaluator - demonstrates the case when we implement transformation class "from scratch". The appropriate class type is rather cumbersome; fortunately, the framework provides us some empty virtual class to inherit from:

```
class eval fself = object
  inherit [string -> int, int] expr_t
  method c_Var s _ x = s x
  method c_Int _ _ i = i
  method c_Add s _ x y = (fself s x) + (fself s y)
  method c_Mul s _ x y = (fself s x) * (fself s y)
end
```

Since we develop a new transformation, we have to take care of types for inherited and synthesized attributes (when we're extending the existing classes these types are already taken care of). Since our evaluator needs a state to bind variables, the type of inherited attribute is `string -> int` and the type of synthesized attribute is just `int`. The implementations of methods are straightforward.

As a final example we consider expression simplification. This time we can make use of plugin "`map`", which in default implementation just copies the data structure (beware: multiplying shared substructures):

```
@type expr =
  Add of expr * expr
| Mul of expr * expr
| Int of GT.int
| Var of GT.string with show, foldl, map
```

In the first iteration we simplify additions by performing constant calculations; we also "normalize" additions in such a way, that if it has one constant operand, then this operand occupies "left" position. The normalization makes it possible to take into account the associativity of addition:

```
class simplify_add =
  let (+) a b =
    match a, b with
    | Int a, Int b -> Int (a+b)
    | Int a, Add (Int b, c)
    | Add (Int a, c), Int b -> Add (Int (a+b), c)
    | Add (Int a, c), Add (Int b, d) -> Add (Int (a+b), Add (c, d))
    | _, Int _ -> Add (b, a)
    | _ -> Add (a, b)
    in
    object inherit @expr[map]
      method c_Add _ _ x y = x.GT.fx () + y.GT.fx ()
    end
```

As we can see, we again concentrated only on the "interesting case"; the implementation of infix "+" may look cumbersome, but this is an essential part of the transformation.

Equally, we can handle the simplification of multiplication:

```
class simplify_mul =
  let ( * ) a b =
    match a, b with
    | Int a, Int b -> Int (a*b)
    | Int a, Mul (Int b, c)
    | Mul (Int a, c), Int b -> Mul (Int (a*b), c)
    | Mul (Int a, c), Mul (Int b, d) -> Mul (Int (a*b), Add (c, d))
    | _, Int _ -> Mul (b, a)
    | _ -> Mul (a, b)
  in
  object
    inherit simplify_add
    method c_Mul _ _ x y = x.GT.fx () * y.GT.fx ()
  end
```

The class `simplify_mul` implements a decent simplifier; however, it overlooks the following equalities: "0*x=0", "0+x=x", and "1*x=x". These cases can be easily integrated into existing implementation:

```
class simplify_all = object
    inherit simplify_mul as super
    method c_Add i it x y =
      match super#c_Add i it x y with
      | Add (Int 0, a) -> a
      | x -> x
    method c_Mul i it x y =
      match super#c_Mul i it x y with
      | Mul (Int 1, a) -> a
      | Mul (Int 0, _) -> Int 0
      | x -> x
    end
```

The interesting part of this implementation is an explicit utilization of a superclass' methods. It may looks at first glance that we handle only top-level case; however, due to late binding, for example, `x.GT.fx ()` in `simplify_mul` implementation is bound to the overriden transformation, which is (in this particular case) is `simplify_all`.

The complete example can be found in file `sample/expr.ml`.

CHAPTER 7

Limitations

Known to be not supported or not taken to account:

- non-regular recursive types
- GADTs

CHAPTER 8

TODOs

Can be a bug:

- Method `on_record_declaration` doesn't introduce new pattern names systematically
- For `compare` and `eq` plugins in case of ADT with single constructor we generate unreachable pattern matching pattern that gives a warning.

Improvements:

- Documentation for `src/GT.ml` is not generated (possible because of a macro).
- Better signature for `method virtual on_record_constr`.
- Custom transformation functions for type parameters has become broken after introducing combinatorial interface for type abbreviations.
- Sometimes we need override class definition for a plugin. It should be possible to specify new custom class inside the attribute.

CHAPTER 9

References

- Dmitry Boulytchev. Code Reuse with Object-Encoded Transformers) // A talk at the International Symposium on Trends in Functional Programming, 2014.
- Dmitry Boulytchev. Code Reuse with Transformation Objects // unpublished.
- Dmitry Boulytchev. Combinators and Type-Driven Transformers in Objective Caml // submitted to the Science of Computer Programming.