GADTs and exhaustiveness: looking for the impossible

Jacques Garrigue
Nagoya University

Jacques Le Normand
System Station

ML Family Workshop 2015

Generalized Algebraic Datatypes

- Algebraic datatypes allowing different type parameters for different cases.
- Similar to inductive types of Coq *et al.*

```
type _ expr =
  | Int : int -> int expr
  | Add : (int -> int -> int) expr
  | App : ('a -> 'b) expr * 'a expr -> 'b expr

App (Add, Int 3) : (int -> int) expr
```

- Able to express invariants and proofs
- Also provide existential types: `∃'a.('a -> 'b) expr * 'a expr`
- Available in Haskell since 2005, and in OCaml since 2012.

GADTs and pattern-matching

- Matching on a constructor introduces local equations.
- These equations can be used in the body of the case.
- The parameter must be a rigid type variable.
- Existentials and unification introduce fresh rigid type variables.

```
let rec eval : type a. a expr -> a = function
  | Int n -> n                                    (* a = int *)
  | Add -> (+)                       (* a = int -> int -> int *)
  | App (f, x) -> eval f (eval x)    (* polymorphic recursion *)
                                    (* ∃b, f : b -> a ∧ x : b *)
val eval : 'a expr -> 'a = <fun>

eval (App (App (Add, Int 3), Int 4));;
- : int = 7
```

GADTs and exhaustiveness

Usually, exhaustiveness is just checking that for each constructor appearing in a pattern-matching, all the other constructors belonging to the same datatype are also matched.

In the case of GADTs, types can be used to refine that.

```
type _ t =
  | Int : int t
  | Bool : bool t

let g : int t -> int = function
  | Int -> 1
```

Here `g` only considers the case `Int`, but this is fine as the case `Bool` would not be allowed by the typing.

Subtleties of exhaustiveness

In some cases, the way to use the typing information may be more subtle.

```
let h : type a. a t -> a t -> bool =
  fun x y -> match x, y with
  | Int, Int -> true
  | Bool, Bool -> true
```

In `h`, the 2 arguments must have the same type, so that `Int, Bool` and `Bool, Int` are impossible. OCaml detects it.

```
let deep : char t option -> char =
  function None -> 'c'
```

Since there exists no value `v` of type `char t`, `Some v` cannot occur either. OCaml didn't detect that.

Asymmetry

```
type zero = Zero and 'a succ = Succ

type (+_,_) vec =
  | Nil : ('a, zero) vec
  | Cons :  'a * ('a,'n) vec -> ('a,'n succ) vec

type _ var =
  | O : 'm succ var
  | S : 'm var -> 'm succ var

let rec lookup : type a n. (a,n) vec -> n var -> a = fun s v ->
  match v, s with
  | O,   Cons (a, _) -> a
  | S m, Cons (_, s) -> lookup s m
val lookup : ('a, 'n) vec -> 'n var -> 'a = <fun>
```

Asymmetry

```
type zero = Zero and 'a succ = Succ

type (+_,_) vec =
  | Nil : ('a, zero) vec
  | Cons :  'a * ('a,'n) vec -> ('a,'n succ) vec

type _ var =
  | O : 'm succ var
  | S : 'm var -> 'm succ var

let rec lookup : type a n. (a,n) vec -> n var -> a = fun s v ->
  match s, v with
  | Cons (a, _), O   -> a
  | Cons (_, s), S m -> lookup s m
Warning 8: this pattern-matching is not exhaustive.
```

OCaml's approach to exhaustiveness

Since the beginning, OCaml's exhaustiveness checker was modified to handle GADTs.

- The original algorithm produced 1 counter-example for non-exhaustive pattern matchings.
- It was modified to produce a set of non-overlapping counter-examples, containing all the missing cases.
- These counter-examples are fed back to the type checker to see whether they are possible or not. *(Haskell 8 shall be similar)*

Since counter-examples use only datatypes appearing in the pattern-matching, and `deep` contains no GADT constructor, its exhaustiveness could not be seen.

Soundness and abstraction

While not complete, we at least expect this approach to be sound.
Paramount, since OCaml uses exhaustiveness for optimizations.

However, we must be careful about abstraction (and injectivity).

```
type (_, _) cmp =
  | Eq : ('a, 'a) cmp
  | Any: ('a, 'b) cmp
module A : sig type a type b val eq : (a, b) cmp end
  = struct type a type b = a let eq = Eq end
let f : (A.a, A.b) cmp -> unit = function Any -> ()
```

Outside of `A`, `A.a` and `A.b` are incompatible. So `Eq` seems impossible.
However `A` itself exports an `Eq` of type `(A.a, A.b) cmp`.

In OCaml, this is avoided by using a type incompatibility relation rather than type equality when typing patterns.

The essence of GADT exhaustiveness

- As we have seen with `deep`, when checking for GADT exhaustiveness, we have to decide whether a given type is inhabited or not.

- Conversely, if we were able to do that, then we could check for exhaustiveness in a complete way (modulo abstraction).

- Unfortunately, even if we limit ourselves to terms built from datatype constructuctors, the inhabitedness is undecidable.

- We can see that by remarking that GADTs exactly allow one to encode Horn clauses.

Encoding of Horn clauses

Horn clauses, the basis of Prolog, are logical clauses of the form:

$$p(\vec{t}) \Leftarrow q_1(\vec{u}_1) \wedge \ldots \wedge q_n(\vec{u}_n)$$

where p and q_i are predicate symbols of fixed arity, and $\vec{t}$ and $\vec{u}_i$ denote sequences of terms (possibly containing term variables).

A set of Horn clauses can be converted into a set of GADT definitions of the form:

$$\texttt{type}\ (\vec{\_})\ p = P : (\vec{u}_1)\ q_1 \times \ldots \times (\vec{u}_n)\ q_n \rightarrow (\vec{t})\ p$$

If there are several clauses with p as conclusion, then they become different branches of the same GADT definition.

Semi-decidability of non-exhaustiveness

From the properties of Horn clauses, we know that

- Provability of a given goal $p(\vec{u})$ is semi-decidable (*i.e.* if there is a proof it will eventually be found, but if there is no proof, proof search may go on forever).

- As a result, the existence of a finite (non-recursive) term of type $(\vec{u})\ p$ is semi-decidable*.

- Since exhaustiveness is about the non-existence of such a term, it is undecidable.

*However, OCaml does allow recursive terms.

Looking for the impossible

While we cannot be complete, bounded proof search provides a sound solution for exhaustiveness. ⇒ Prolog's SLD-resolution

To implement it, we converted the function type checking patterns to CPS. This allows us to backtrack, enumerating counter examples.

Starting from the counter-example found by the (untyped) exhaustiveness analysis, which contains or-patterns, we run the type-checking function, with the following changes.

- When reaching an or-pattern node, save the state, and try all the branches until finding a successful one, backtracking between them.

- When reaching a wild-card node, if its inferred type is a GADT, and if this is the first one from the root of the pattern, explode it into an or-pattern of all its cases.

Typing and exploding using CPS

```
(* mode is Check or Type, k is the continuation *)
let rec type_pat mode env spat expected_ty k =
  match spat.ppat_desc with
  | Ppat_any -> (* wild card *)
      if mode = Check && is_gadt expected_ty then
        type_pat mode env (explode_pat !env expected_ty) expected_ty k
      else k (mkpat Tpat_any expected_ty)
  | Ppat_or (sp1, sp2) -> (* or pattern *)
      if mode = Check then
        let state = save_state env in
        try type_pat sp1 expected_ty k
        with exn ->
          set_state state env;
          type_pat sp2 expected_ty k
      else
      (* old code *)
  | Ppat_pair (sp1, sp2) -> (* pair pattern *)
      let ty1, ty2 = filter_pair env expected_ty in
      type_pat mode env sp1 ty1 (fun p1 ->
      type_pat mode env sp2 ty2 (fun p2 ->
        k (mkpat (Tpat_pair (p1,p2) expected_ty))))
  | ... (* other cases in CPS *)
```

Examples

```
type zero = Zero and _ succ = Succ
type (_,_,_) plus =
  | Plus0 : (zero, 'a, 'a) plus
  | PlusS : ('a, 'b, 'c) plus -> ('a succ, 'b, 'c succ) plus

let trivial : (zero succ, zero, zero) plus option -> bool
  = function None -> false                                  (* ok *)
let easy : (zero, zero succ, zero) plus option -> bool
  = function None -> false                                  (* ok *)
let harder : (zero succ, zero succ, zero succ) plus option  -> bool
  = function None -> false  (* fails, requires deeper exploding *)
let inv_zero : type a b c d. (a,b,c) plus -> (c,d,zero) plus -> bool
  = fun p1 p2 -> match p1, p2 with Plus0, Plus0 -> true
         (* ok, thanks to exploding of second _ in (PlusS _, _) *)
```

Unused cases

Using the same approach we can also accurately check for unused cases.

Ideally, should be symmetric with exhaustiveness (done yesterday!)

```
let h : type a. a t -> a t -> bool =
  fun x y -> match x, y with
  | Int, Int -> true
  | Bool, Bool -> true
  | _ -> false ;;
Warning 11: this match case is unused.
```

Must be checked for each line, so it can be expensive.

Issues

We have just implemented a heuristics. A number of issues remain.

- How deep should we go? Our heuristics seems good enough, but it fails in some cases, and may exhibit an exponential behavior in others. We see 2 alternative solutions.
  - Use a special syntax for empty patterns, like in Agda. This avoids always running the refutation, but can be heavy.
  - Add a parameter to pattern-matching, indicating the depth of the search.
- Shall we also explode non-GADT types? At least when there is only one case?
- What should we do about redundancy?
  Maintaining the symmetry appears expensive.

Benchmarks

	camlinternalFormat	stdlib	make all
Exhaustiveness	3.1s	11.4s	100s
Exhaust. & Unused	3.4s	11.8s	102s
Exhaust. w/o explode	3.9s	12.2s	101s
Exh. & Un. w/o expl.	4.5s	13.0s	103s

Conclusion

– We have implemented a strengthened exhaustiveness and redundancy check for GADTs in OCaml.

– It is not complete, since this is undecidable, but does a bounded proof search. This helps in concrete cases, and avoids some asymmetrical behaviors.

– The code is at*

```
svn co http://caml.inria.fr/svn/ocaml/branches/gadt-warnings
```

– We need your feedback about concrete examples.

*Code URL in abstract is wrong.

Logic using GADTs

```
type falso
let ex_falso (_ : falso) = assert false
type 'p not = 'p -> falso
type ('a,'b) ior = Inl of 'a | Inr of 'b
(* NB: the unit -> below is essential for soundness *)
type classic = {classic: 'p. unit -> ('p, 'p not) ior}
type (_,_) eq = Refl : ('x,'x) eq

(* Drinkers paradox: exists x, x drinks -> forall y, y drinks *)
type _ drinks
type all_drink = {all: 'y. unit -> 'y drinks}
type non_drinker = Nd : 'x drinks not -> non_drinker
type drinkers_paradox = Drinker : ('x drinks -> all_drink) -> drinkers_paradox
let proof {classic=c} : drinkers_paradox =
  match c () with
  | Inl (Nd nd) -> Drinker (fun d -> ex_falso (nd d))
  | Inr nnd -> Drinker (fun d ->
      {all = fun () -> match c() with
      | Inr nd -> ex_falso (nnd (Nd nd))
      | Inl d -> d})
```