GADTs and exhaustiveness: looking for the impossible

Jacques Garrigue and Jacques Le Normand

1 Synopsys

Sound exhaustiveness checking of pattern-matching is an essential feature of GADTs, and OCaml has supported it from day one, by showing that the remaining cases could never be typed [1]. Not only does it allow the programmer to be confident in the soundness of his code, but it also permits optimizations which make GADTs more efficient. However, while this approach is sound and can prune some simple uses of GADTs, some other uses caused superfluous warnings. In this talk we describe the original approach and how we ensure its soundness, and show that one can do better by turning the type-checking of extra cases into a backtracking proof search algorithm. We also show that the exhaustiveness problem is undecidable for GADTs, so that this proof search must be kept partial.

2 GADTs and exhaustiveness

Checking the exhaustiveness of pattern-matching is a difficult problem. Technically, it is about checking whether there are values of the matched type that are not covered by the cases of the pattern-matching. There are well-known techniques to handle this problem for algebraic datatypes [3], but they do not attempt to tackle semantical questions, such as whether such a value can be built or not. For instance consider the following example:

```
type empty = {e : 'a. 'a}
let f : empty option -> unit = function None -> ()
```

Since there is no way to build a value of type `empty`, this match is actually exhaustive, but the checker will still report a missing `Some` case.

For normal types, this limitation does not matter: why would one intentionally introduce an empty type? However, in the case of GADTs, the problem becomes acute.

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

let f : type a. a t -> a = function
  | Int -> 1
  | Bool -> true

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

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

Function `f` is a classical GADT function, where different branches instantiate the type parameter differently. It is clearly exhaustive. If we only look at the constructors, the function `g` is not exhaustive. However, its input type is restricted to `int t`, which is incompatible with the constructor `Bool`, so that the only valid input is `Int`, making it exhaustive. Function `h`, is also exaustive, because required `x` and `y` to have the same type. These functions have useful instances, and we want them to be recognized as exhaustive.

3 First implementation

We were in a conundrum: a complete exhaustiveness check seemed very difficult[1], yet we wanted to add GADTs to the OCaml compiler and we needed a simple way to check the exhaustiveness. Our initial algorithm simply took all the missing patterns from the exhaustiveness checker, and type checked them one by one in order to see if they were actually possible patterns. Unfortunately the original exhaustiveness algorithm did not return a complete enough set of patterns. For example, in the previous example `h`, the exhaustiveness checker would only return `Int, Bool` as a missing pattern, while we also needed to check that `Bool, Int` is an invalid pattern to remove any possible doubt that `h` is actually exhaustive. Consequently, we modified the exhaustiveness checker so that it would return a complete set of missing patterns. However, as we will see, enthusiastic GADT users were more clever than the checker, and they got exhaustiveness warnings where none should be [5].

4 Abstract types

To make matters worse, in OCaml it is impossible to know if two abstract types exported from other modules are equal or not. Take, for example:

```
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 -> ()
```

This program properly signals that the function f is non exhaustive. Indeed, even though the types `A.a` and `A.b` appear to be different outside of module `A`, they are in fact the same. To handle these kinds of cases, a new compatibility relation is introduced, and when the type checkers

[1] It turned out to be undecidable.

tries to unify indices of GADTs during pattern typing, it refers to this relation for non-unifiable type constructors, rather than immediately raising a unification error. In particular this compatibility relation assumes that abstract types are compatible with all other types. In this particular case, when typing the missing case `Eq`, it simply assumes `A.a` and `A.b` are compatible. In other words, the type checker is far more permissive with GADT indices inside patterns than inside expressions. In doubt, it is better to permit possibly impossible patterns and to reject potentially unsafe expressions. Note that since we use exactly the same function to type-check patterns and to check exhaustiveness, if the exhaustiveness check reports a missing pattern, then type checking will always allow it[2].

5 Exploding and backtracking

While our original approach seemed mostly satisfactory, there are cases where it fails. For instance, consider the following function:

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

Since `t` is only defined for `int` and `bool`, `char t` is actually the empty type, *i.e.* there are no values of the form `Some _` at type `char t option`. However, to see that one needs to explode `_` into its different cases, and check them separately. This gives us the following two patterns:

```
Some Int
Some Bool
```

Then we can call the type checker as before, to verify that they are incompatible with the given type.

Note that as soon as we start to do deeper case analysis, the approach switches from just checking whether a pattern is type to checking whether a particular type is inhabited by terms of a certain form. Here are a few more examples of the same kind, by order of difficulty.

```
type zero = Zero
type _ 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
let easy :
  (zero, zero succ, zero) plus option -> bool
  = function None -> false
let harder :
  (zero succ, zero succ, zero succ) plus option
    -> bool
  = function None -> false
```

`zero` and `succ` encode type level natural numbers. `plus` is the Peano version of addition, in relational form; namely there is a term (a, b, c) `plus` if and only if $a + b = c$. `trivial` can be easily checked, as `(zero succ, zero, zero)` does not match either of `Plus0` and `PlusS`. `easy` is a bit more difficult, as it seems to match `Plus0`, but unification between `zero succ` and `zero` fails later. For `harder`, unification with `PlusS` succeeds, however the argument becomes `(zero, zero succ, zero) plus`, which was inferred empty in `easy`.

In `deep`, `trivial` and `easy` it is sufficient to explode the first `_` according to its inferred type. However, `harder` requires to infer the type of the argument of the GADT constructor `PlusS` in order to explode it once more.

Another interesting case is when there is a dependency between components of a tuple.

```
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
```

Here the extra patterns coming from the basic exhaustiveness algorithm are:

```
Plus0, PlusS _
PlusS _, _
```

While the first pattern is clearly empty, the second one is typable if one does not explode the second `_`. However, to do that we would need to first infer the type of the second component of the pair, which depends on the freshly generated first component. In this case again, typing patterns (for checking emptyness) and exploding wildcards must be interleaved.

The solution to this conundrum is to actually do all of these simultaneously. Namely, we modified the recursive `type_pat`[3], which is the main function for typechecking patterns, in order to turn it into a proof-searching function. The basic idea is to make it non-deterministic. However, since this function uses side-effecting unification, returning a list of results would not be easy. Rather we converted to continuation passing style, using backtracking to cancel unification where needed. In particular, it is sufficient to explode wild cards into or-patterns, as they are then interpreted in a non-deterministic way, allowing to check all combinations.

```
(* 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
```

[2]For a long time this was not the case with GHC. But there is some progress in the Haskell world too [2].

[3]The code is available through OCaml's Subversion server: `svn co http://caml.inria.fr/svn/ocaml/branches/gadt-warnings`.

```
  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 *)
```

6 Undecidability and heuristics

The above definition of `type_pat`, in all its expressiveness power, gives also a strong hint at why exhaustiveness checking of GADTs in undecidable. A simple way to see it is that GADTs can encode Horn clauses in a very direct way, each type definition being a predicate, and each constructor a clause, with its arguments the premises. Then the `type_pat` functions precisely implements Prolog's SLD resolution, for which counter-example generation (*i.e.* construction of a witness term) is known to be only semi-decidable. Another way to see it is that one can encode execution traces of some arbitrary Turing machine in a GADT definition, so that exhaustiveness checking is equivalent to the halting problem.

This undecidability means that we have to find a good heuristics as to where to abandon the search. Note that the complexity is exponential in the number of wildcard patterns exploded. A simple heuristics, that seems sufficient in most cases, is to only explode wildcard patterns which have only GADT constructors and do not explode any of the generated subpatterns. This means that the `harder` example above would be flagged non-exhaustive while all the other examples would be correctly identified as exhaustive. Here is another example which would be incorrectly flagged:

```
let deeper : (char t * bool) option -> char =
  function None -> 'c'
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
Some _
```

Here the wild card corresponds to a tuple type, so that the case-analysis would stop there. Even in this very limited approach, one can still exhibit an exponential behavior:

```
type _ t =
  A : int t | B : bool t | C : char t | D : float t
type (_,_,_,_) u = U : (int, int, int, int) u
let f : type a b c d e f g h.
   a t * b t * c t * d t * e t * f t * g t * h t
       * (a,b,c,d) u * (e,f,g,h) u -> unit =
  function A, A, A, A, A, A, A, A, U, U -> ()
```

The above check takes about 10 seconds to exhaust all 65536 cases. As in Prolog, one can dramatically improve performance by changing the pattern order.

Independently of the heuristics chosen, there will always be cases where one would like the algorithm to try harder. We can think of at least two ways to handle those. One is to introduce an absurd pattern *à la* Agda [4]. This pattern would tell the checker to try hard to prove emptyness at this particular position. However, this also requires introducing cases without right-hand side at the syntactic level. Another approach would be to use attributes on the `match` and `function` constructs to indicate how hard we want the checker to try: `function [@exhaust 10] None -> ()`. In this case we would need a precise definition of the strength of the search.

7 Unused cases

The dual of exhaustiveness checks is the detection of unused cases. Take, for example:

```
let deep' : char t option -> char = function
  | None -> 'c'
  | Some _ -> 'd'
```

Since we added a pattern at the end of an already exhaustive match, it is clearly redundant.

The approach is similar: after refining the pattern to keep only subcases that are not covered by previous cases, one must check whether they are inhabited or not. Currently this second check is not done; doing it would require making the redundancy algorithm return an explicit list of cases. While detecting ususued cases is technically less important —there is no direct impact on soundness for instance—, having accurate warnings would help the programmer reason about his program. Note however that we cannot hope to detect all unused cases, in the same way that we cannot guarantee that all counter-examples of exhaustiveness are really inhabited.

References

[1] Jacques Garrigue and Jacques Le Normand. Adding GADTs to OCaml: the direct approach. In *OCaml Meeting*, September 2011.

[2] Georgios Karachalias, Tom Schrijvers, Dimitrios Vytiniotis, and Simon Peyton Jones. GADTs meet their match. In *ICFP*, 2015.

[3] Luc Maranget. Les avertissements du filtrage. In *Journées Francophones des Langages Applicatifs*, 2003.

[4] Ulf Norell. Dependently typed programming in Agda. In *AFP 2008*, volume 5832 of *Springer LNCS*, pages 230–266, 2009.

[5] GADT exhaustiveness check incompleteness. OCaml problem report #6437, May 2014. `http://caml.inria.fr/mantis/view.php?id=6437`.