GADTs and Exhaustiveness: Looking for the Impossible

Jacques Garrigue
Nagoya University Graduate School of Mathematics
garrigue@math.nagoya-u.ac.jp

Jacques Le Normand
rathereasy@gmail.com

Sound exhaustiveness checking of pattern-matching is an essential feature of functional programming languages, and OCaml supports it for GADTs. However this check is incomplete, in that it may fail to detect that a pattern can match no concrete value. In this paper we show that this problem is actually undecidable, but that we can strengthen the exhaustiveness and redundancy checks so that they cover more practical cases. The new algorithm relies on a clever modification of type inference for patterns.

1 Introduction

Exhaustiveness and redundancy checks of pattern-matching are important features of functional programming languages [11]. They let programmers write and maintain software more efficiently and with fewer bugs. In OCaml, they even allow extra optimizations in the pattern-matching compiler [9][1].

In OCaml 4.00 [10], pattern-matching was extended to accommodate GADTs [6]. GADT is a shorthand for Generalized Algebraic Data Types [1, 20, 2] which denotes an extension of Algebraic Data Types where the definition may contain type equalities specific to a branch. This extension clearly has an impact on exhaustiveness and redundancy checking. However, while the question of exhaustiveness of pattern-matching has been studied extensively in the context of dependent type systems [3, 12, 19, 17, 4, 13, 15], the more restricted case of GADTs used in practical functional programming languages was not well explored. For instance GHC, which already had GADTs since 2005, didn't support them in either check [16]. Ironically, even Xi, who had already shown how types could be used to prune some cases in his work on Dependent ML [19], didn't try to use that in the context of GADTs [20]. One reason might be that the coverage check for dependently typed pattern-matching is known to be undecidable in general [12], and as such the road was potentially slippery.

For this same reason, we settled for a minimal modification of the exhaustiveness check, which had to be clearly sound, but didn't attempt to be complete. The basic idea was to collect the missing cases from a pattern-matching, and then use type inference to prune the impossible ones. While this limited modification already proved to be ambitious, as was shown by a number of bugs in corner cases that had to be fixed, it also appeared to be insufficient, as some users encountered spurious warnings on code whose exhaustiveness was clear enough. Moreover, modifying only the exhaustiveness check created an asymmetry with the redundancy check.

In this paper we both describe the original approach, including what we had to do to ensure its soundness, and a new improved algorithm, which supports both exhaustiveness and redundancy checks, and is able to prove exhaustiveness in more cases. The basic idea behind this new algorithm is to see both checks as trying to prove that a type is not inhabited, which can be done by turning type inference into a proof search algorithm. Unfortunately, a quick look a the search space, which exactly matches resolution

[1] The propagation of exhaustiveness information to pattern matching compilation in OCaml was originally introduced by Jacques Garrigue, to allow better compilation of polymorphic variants [5].

Jeremy Yallop and Damien Doligez (Eds.)
ML/OCaml 2015
EPTCS 241, 2017, pp. 23–35, doi:10.4204/EPTCS.241.2

for Horn clauses, is enough to show that the exhaustiveness and redundancy problems are undecidable even when limited to GADTs. This means that this proof search must be restricted so that its termination can be enforced in a predictable way.

The idea of using a standard non-determinism technique to turn type inference into a proof search applicable to exhaustiveness and redundancy checks is the main contribution of this paper. We complete it with some basic heuristics, and syntactic additions to the language that the programmer can use to provide some guidance to the exhaustiveness checker in its proof search.

This paper is organized as follows. In section 2, we explain what is the exhaustiveness problem for GADTs through examples. In section 3, we describe the original implementation, and how we needed to introduce a new notion of type compatibility to make it sound. In section 4 we describe the new proof search algorithm. In section 5 we show that the problem is undecidable for GADTs, and discuss practical heuristics and language features to accommodate this. Finally, in section 6, we show how the redundancy check can be integrated in this picture, and describe the concrete semantics implemented in OCaml 4.03.

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 data types [11], 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` (at least in a type-safe call-by-value language), 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 g1 : type a. a t -> a = function
  | Int -> 1
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
Bool

let g2 : 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 just remove the second case without changing the type annotation, in function `g1`, the pattern-matching becomes incomplete, and a missing case `Bool` should be reported. However, in function `g2`, we change the type annotation to restrict its input to be of type `int t`, which is incompatible with the constructor `Bool`, so that the only valid input is `Int`, making it exhaustive. Function `h`, is also exhaustive, because it requires `x` and `y` to have the same type. Examples `g2` and `h` have useful instances, and we want them to be recognized as exhaustive, but this requires the exhaustiveness check to take type information into account.

3 First implementation

When we first implemented type inference for GADTs in OCaml [6], we did not know for certain that the exhaustiveness problem was undecidable, but it seemed highly unlikely that there was a simple and complete algorithm to solve it. Our strategy was to find a simple conservative algorithm that was easily understandable and that would catch all the potential bugs for the programmer.

3.1 Algorithm

Our first algorithm took the approach of using the original exhaustiveness check to extract a list of missing patterns, and then rely on the type checker to prove that they are not inhabited. For this, we introduced the following two changes to the exhaustiveness checker.

1. The original exhaustiveness algorithm did return a counter example when a pattern-matching was not complete, but this example did not cover all the missing cases. For instance, 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.
2. Each pattern in the missing set is then fed to the type checker, to detect whether it is typeable or not. Untypeable patterns cannot be matched by any value, so it is safe to ignore them. If no pattern remains, the pattern-matching is deemed exhaustive, otherwise the or-pattern of all the remaining patterns is returned as an exhaustive counter-example. Note that when typing patterns, one needs to relax unification, as we will see in the next subsection.

3.2 Incompleteness

This approach seemed sufficient at first, as almost all exhaustive pattern-matchings were detected as such. However, enthusiastic GADT users were more clever than the checker, and they got exhaustiveness warnings where none should be [18]. The problem there involved using variables from a finite set to access a finite length context, but the essential cause can be seen in the following example.

```
type _ is_int = IsInt : int is_int

let h2 : type a. a t * a is_int -> bool = function
  | Int, IsInt -> true
```

The generated counter-example here is `(Bool, _)`. As this pattern appears to be typeable, a warning was emitted. This warning is of course spurious, since the only value which can replace the wildcard

$$\vdash_E \alpha \simeq \beta \qquad \frac{\text{type } (\vec{\alpha})t \in E \quad \tau' \neq (\vec{\theta}')t}{\vdash_E (\vec{\tau})t = \tau'} \qquad \frac{\vdash_E \tau_i \simeq \tau_i' \ \ (i \in \mathrm{Inj}_E(t))}{\vdash_E (\tau_1,\ldots,\tau_n)t \simeq (\tau_1',\ldots,\tau_n')t}$$

$$\vdash_E \tau \simeq \tau \qquad \frac{\vdash_E \tau_1 \simeq \tau_2}{\vdash_E \tau_2 \simeq \tau_1} \qquad \frac{\vdash_E \tau_1 \simeq \tau_1' \quad \vdash_E \tau_2 \simeq \tau_2'}{\vdash_E \tau_1 \to \tau_2 \simeq \tau_1' \to \tau_2'} \qquad \frac{\vdash_E \tau_1 \simeq \tau_1' \quad \vdash_E \tau_2 \simeq \tau_2'}{\vdash_E \tau_1 \times \tau_2 \simeq \tau_1' \times \tau_2'}$$

$$\frac{\begin{array}{c}\text{type } (\vec{\alpha})\ t = C_1 \text{ of } \theta_1 \mid \ldots \mid C_n \text{ of } \theta_n \quad \text{type } (\vec{\alpha}')\ t' = C_1 \text{ of } \theta_1' \mid \ldots \mid C_n \text{ of } \theta_n' \\ t \neq t' \quad \vdash_E \theta_i \simeq \theta_i' \ \ (1 \leq i \leq n) \quad \vdash_E \tau_i \simeq \tau_i'\end{array}}{\vdash_E (\tau_1,\ldots,\tau_n)t \simeq (\tau_1',\ldots,\tau_n')t'}$$

Figure 1: Compatibility relation

is `IsInt`, and its type would be incompatible with `Bool`. We will see later that this replacement of wildcards by the concrete cases from the type definition can be done systematically. As a side remark, one may notice that reversing the components of the pair in this example makes exhaustiveness detection easier: then the only counter-example becomes `(IsInt, Bool)`, which is clearly impossible. However, this is no sufficient reason to just ignore the problem.

3.3 Type compatibility

While reusing the type checker to detect non matchable patterns seems a good idea, it relies on the following fact:

Fact 1 *If a pattern p matches a value v, and v can be given some type τ, and p does not contain or-patterns, then p can also be given type τ.*

A similar lemma appears in [19] for instance. This seems reasonable: a pattern is built like a value, except that some parts may be wild cards, which can be replaced by any value of the same type, so that this fact can be proved by an easy induction.

However, one must be careful that typeability in ML is a function of the context. Concretely, the ML module system allows one to hide type equalities, so that it is for instance 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 -> ()
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
Eq
```

Since there is no relation between `a` and `b` in the signature of `A`, `Eq` cannot be given the type `(A.a, A.b) cmp` outside of module `A`, which might lead one to think that the function `f` is exhaustive. However, inside `A` we can use the type equality `a = b` to create such a value and export it.

The exhaustiveness checker should detect that `f` is not exhaustive. Of course, it cannot do it by searching the whole typing environment for a way to build such a value, as this would clearly be undecidable, and we need a conservative answer. Rather, we change the definition of typeability for patterns, so that `Eq`, as a pattern, can be given the type `(A.a, A.b) cmp`.

In general, this means that we allow typing patterns assuming extra type equalities for abstract types. Technically, a new compatibility relation on types is introduced, and when the type checker unifies indices of GADTs (the type parameters of a GADT constructor) during the typing of patterns, it refers to this relation for non-unifiable type constructors appearing inside these indices, rather than immediately raising a unification error. The compatibility relation (for only some features of the language) is the smallest relation including the rules of Figure 1. For simplicity, this relation doesn't try to be too restrictive, and for instance it doesn't distinguish between different type variables. More importantly, the second rule expresses our assumption that abstract types are compatible with all other types. As a result of these 2 rules, this relation is reflexive and symmetric, but not transitive. Additionally, the next rule, concerning identical type constructors, says that we shall only check compatibility for injective parameters, *i.e.* parameters whose equality can be deduced from the equality of the resulting types. Finally the last rule says that two data types are compatible if their data constructors have the same names (and order), and the types of their arguments are compatible.

In our example above, when typing the missing case `Eq`, the compatibility relation says that `A.a` and `A.b` are compatible. This compatibility relation makes the type checker far more permissive with GADT indices inside patterns than inside expressions. This is fine because, in doubt, it is safe to allow 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, we have the nice property that if the exhaustiveness check reports a missing pattern, then type checking will always allow it. This is to contrast with what used to happen in GHC before version 8.0, as the exhaustiveness checker could tell you that some case was missing, but adding it would cause a type error, leaving the programmer with the only option of adding a catch all case. Thanks to recent work [7], GHC 8.0 has the same property, by using the same typechecking-based pruning; note that since Haskell does not allow the same kind of type abstraction as ML, they do not need our compatibility relation.

4 Splitting and backtracking

While our original approach seemed mostly satisfactory, there are cases where it fails. We have already seen such an example in Section 3.2. Here is an even simpler one.

```
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, there is no order to change, and to see that `char t` is empty one needs to split `_` 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.

We just happened to rediscover in a very natural way the idea of *context splitting*, originally introduced by Coquand for doing pattern matching on dependent types [3].

Note that as soon as we start to do deeper case analysis, the approach switches from just checking whether a pattern has some 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 split 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 split 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 typeable if one does not split 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 emptiness) and splitting wildcards must be interleaved. From a more theoretical point of view, this is the reason why Coquand used the name *context splitting*, rather than just *pattern splitting*. Previous choices change the splitting we can do next.

The solution to this inter-dependence is to actually do all of these simultaneously. Namely, we modified the recursive `type_pat`, 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 it to continuation passing style, using backtracking to cancel unification where needed. In particular, it is sufficient to split wild cards into or-patterns[2], as they are then interpreted in a non-deterministic way,

[2] *or-patterns* are patterns of the form $pat_1 \mid pat_2$, which catch the union of pat_1 and pat_2. Since we are splitting wild cards, in this case they do not contain variables.

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 mode env sp1 expected_ty k
      with exn ->
        set_state state env;
        type_pat mode env 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 *)
```

As you can see, this modified `type_pat` function[3] has a special handling for `Ppat_or` in `Check` mode, which behaves as though the whole pattern were duplicated, with the or-pattern replaced by either of its sub-patterns. This means that, rather than calling `type_pat` individually on each missing pattern, it becomes possible to call it directly on an or-pattern combining all the missing cases, and it is fine to leave some or-patterns deep inside. This makes generating the patterns for missing cases simpler, but more importantly this allows to introduce such or-patterns on the fly by splitting wild cards while searching for a typeable case, as you can see in the handling of `Ppat_any`. For the remaining cases, such as `Ppat_pair`, one just has to use continuation passing style, to allow the remainder of the code to be called several times, instantiated with different branches of or-patterns.

One should not confuse this use of type checking inside the exhaustiveness check, with other more theoretical works where the check is integrated into type checking [4, 8]. There might be some interesting connections, but here we are still relying on an independent exhaustiveness analysis to produce the missing patterns.

5 Undecidability and heuristics

The above definition of `type_pat`, in all its expressive power, gives also a strong hint at why exhaustiveness checking of GADTs is 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. For instance the type `plus` defined in Section 4 is actually an encoding of the following Horn clauses, in Prolog syntax:

[3] The code is part of OCaml source code since version 4.03, and can be found in the online repository: `https://github.com/ocaml/ocaml/blob/trunk/typing/typecore.ml`.

```
type s0 and fin                                                    (* states *)
type c0 and blank                                                  (* symbols *)
type left and right                                                (* directions *)
type endt                                                          (* end of tape *)

type ('st_in, 'head_in, 'st_out, 'head_out, 'lr) transition =
  | Tr1 : (s0, blank, s1, c0, left) transition
  | Tr2 : (s1, blank, s0, c0, right) transition
  | Tr3 : ('s, c0, fin, c0, left) transition

type ('state, 'head, 'left, 'right) eval =
  | Tm_fin       : (fin, _, _, _) eval
  | Tm_mv_left   : ('st_in, 'head_in, 'st_out, 'head_out, left) transition *
                   ('st_out, 'head_out, 'left, 'head_out * 'right) eval ->
                   ('st_in, 'head_in, 'head * 'left, 'right) eval
  | Tm_mv_right  : ('st_in, 'head_in, 'st_out, 'head_out, right) transition *
                   ('st_out, 'head_out, 'head_out * 'left, 'right) eval ->
                   ('st_in, 'head_in, 'left, 'head * 'right) eval
  | Tm_ext_left  : ('state, 'head, blank * endt, 'right) eval ->
                   ('state, 'head, endt, 'right) eval
  | Tm_ext_right : ('state, 'head, 'left, blank * endt) eval ->
                   ('state, 'head, 'left, endt) eval

type goal = (s0, blank, endt, endt) eval

let f : goal option -> unit = function None -> ()
```

Figure 2: Encoding of the halting problem

```
plus(zero, X, X).
plus(succ(X), Y, succ(Z)) :- plus(X, Y, Z).
```

Reciprocally, any GADT can be encoded as a set of Horn clauses.

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

would become:

```
int.                                                    (* int is inhabited *)
expr(int) :- int.                                               (* Int case *)
expr(int -> int -> int).                                        (* Add case *)
expr(B) :- expr(A -> B), expr(A).                               (* App case *)
```

Then the `type_pat` function implements Prolog's SLD resolution, for which counter-example generation (*i.e.* construction of a witness term) is known to be at best semi-decidable (since resolution for Horn clauses is only semi-decidable).

Using this view of GADTs as Horn clauses, we can also encode an implementation of Turing machines in Prolog into GADT definitions, so that exhaustiveness checking is equivalent to the halting problem. This encoding is shown in Figure 2, where `transition` encodes the definition of the Turing machine (its valid transitions), and `eval` encodes the execution of this Turing machine, succeeding if

the final state `fin` is reached. The function `f` is exhaustive if and only if the Turing machine terminates starting from state `s0` on an empty tape.

This proof of undecidability is not new: McBride already demonstrated an inductive type encoding traces of Turing machines in [12, Section 6.4], and used it as proof of the undecidability of exhaustiveness. Similarly, Oury gave an even shorter example encoding the Post correspondence problem [15]. In both cases, it is trivial to convert definitions so that they fit into the scope of GADTs. What is newer is the use of Horn clauses to make the connection between a semi-decidable implementation in Prolog and the corresponding GADT definitions.

It is interesting to note that GHC does not have the same undecidability problem, at least if we restrict it to lazy pattern matching. Namely, in GHC every type is inhabited by the value `undefined`.

```
undefined :: forall a. a
undefined = undefined
```

As a result, one only has to check the remaining cases when a constructor is actually matched on, as this forces evaluation, but not for wild cards. This explains why Karachalias *et al.* [7] did not discuss this problem. Note however that, since GHC has strictness annotations, one can actually create situations where inhabitation by constructor terms becomes relevant, and the exhaustiveness check becomes undecidable.

For OCaml, 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 split. A simple heuristics, that seems sufficient in most cases, is to only split wildcard patterns when they do not generate multiple branches (*i.e.* tuple types or data types with a single case), or when they have only GADT constructors (but then do not split 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:

```
type ('a,'b) sum = Inl of 'a | Inr of 'b;;
let deeper : (char t, char t) sum 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 points to a sum type with multiple cases, so that the case-analysis stops 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. An interesting approach is the concept of the *absurd pattern*, introduced by Agda [14]. This pattern is a hint to the checker that no data constructor can come at this position, which can then be proved trivially. In Agda, a match case containing an absurd pattern has no right-hand side.

```
data Fin : Nat -> Set where
  fzero : {n : Nat} -> Fin (suc n)
```

```
  fsuc : {n : Nat} -> Fin n -> Fin (suc n)

magic : {A : Set} -> Fin zero -> A
magic ()
```

In this example, the type `Fin n` denotes natural numbers smaller than `n`, so that there is no value of type `Fin zero`. As a result, one can define a function from `Fin zero` to any type, by matching on the absurd pattern `()`.

While absurd patterns have nice properties, they have also one drawback: in Agda one has to manually split the cases, so that every `()` occurs at a trivially impossible position. This can be pretty verbose.

The solution we implemented in OCaml is a variant of absurd patterns, but where we still rely on proof search to prove emptiness. We call these *refutation cases*, as the syntax is denoted by a single dot "`.`" on the right hand side rather than by a specific pattern. For instance, the `deeper` example above can be written

```
let deeper' : (char t, char t) sum option -> char = function
  | None -> 'c'
  | Some Inl _ -> .
  | Some Inr _ -> .
```

meaning that in the last two cases the right hand side is unreachable. Since it becomes syntactically possible to have a function with no concrete case, we can even write the `magic` function

```
let magic : char t -> 'a = function _ -> .
```

The main difference with the absurd pattern approach is that the absurd positions are automatically inferred. Namely, we first compute the reachable parts of the pattern, by removing the parts matched by previous cases, and then apply the above restricted proof search to check that the remaining pattern is empty. Since one can write more precise patterns, it becomes possible to point down impossible positions explicitly; in practice it suffices to write a wildcard, and the proof search will look for the actual contradiction inside this wildcard. This means that the following (exhaustive) variant of `harder` is very compact:

```
let harder' : (zero succ, zero succ, zero succ) plus -> 'a = function PlusS _ -> .
```

6 Unused cases, refutation cases, and exhaustiveness

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. While detecting unused cases is technically less important —there is no direct impact on soundness for instance— having accurate warnings helps the programmer reason about his program.

One can easily see that unused cases and refutation cases use exactly the same algorithm, *i.e.* if a case can be proved unused, it can be turned into a refutation case. In OCaml 4.03, the above definition causes the following warning:

```
Warning 56: this match case is unreachable.
Consider replacing it with a refutation case '<pat> -> .'
```

Note that we only suggest a refutation case if types are needed to prove emptiness. Of course, in the same way that we cannot guarantee that all counter-examples of exhaustiveness are really inhabited, we cannot hope to detect all unused cases, but at least we have here some form of symmetry.

Another consequence of this symmetry is that refutation cases are actually handled by the redundancy check rather than the exhaustiveness check. Namely, the algorithm implemented in OCaml 4.03 works as follows:

1. For each case, compute its *residual*, *i.e.* the intersection of the pattern with the complement of the previous cases.
2. Check whether this residual is inhabited or not, using `type_pat`, which implements the above heuristics of splitting wild cards only once.
3. If this is a refutation case, and the residual could not be proved empty, emit an error as the refutation failed.
4. If this is a concrete case, and the residual is empty emit a warning as this case is unused.
5. After processing all the cases, compute the or-pattern of the missing cases, which is actually the same as computing the residual of an extra catch-all case.
6. If the pattern-matching contained only one case, use `type_pat` with the same heuristics to check that this final residual is empty. If there are more than one cases, use `type_pat` without splitting. In both cases, emit a non-exhaustiveness warning if the residual cannot be proved empty.

The choice of using a different heuristics for exhaustiveness when there are more than one case may seem surprising, but it appears that there is a significant difference in cost for usual code. Table 1 shows

Exhaustiveness(1/many)	Redundancy	Time(sec)
1/1	1	7.50
1/0	1	7.00
0/0	1	7.00
0/0	0	6.75

Table 1: Compilation times for the standard library

compilation times for OCaml's standard library, which contains both GADT-free code (most modules) and GADT-heavy code (the Format module and its dependencies). Of course, one can reduce cost more by never splitting wild cards, but as discussed above this may make code more verbose. On the other hand, disabling splitting for exhaustiveness alone is less of a problem, as one can always add a plain refutation case "`_ -> .`". The rationale for having a different behavior when there is only one case is that some syntactic forms do not allow to add an extra case, and the extra cost appears to be insignificant. In particular, this allows to prove exhaustiveness for all examples here except `harder` and `deeper`.

7 Conclusion

Exhaustiveness and redundancy checks have been an important part of the OCaml compiler since its inception, but only recently received a major overhaul because of the introduction of GADTs. The initial changes were the smallest changes we could make which accommodated GADTs and consequently the

resulting algorithm had some shortcomings. By discovering the link between exhaustiveness checks and logic programming (Horn clauses), we were able to significantly improve the algorithm and at the same time understand the limits of what we could possibly do. The introduction of refutation cases to overcome these limits actually furthers the duality of exhaustiveness and redundancy checks, to give us a more regular system of warnings.

Acknowledgements

We are grateful for the useful comments provided by anonymous reviewers, and for the feedback from the OCaml community on our experiments. This work was partially supported by JSPS Grant-in-Aid for Scientific Research 16K00095.

References

[1] Lennart Augustsson & Kent Petersson (1994): *Silly Type Families*. Draft.

[2] James Cheney & Ralf Hinze (2003): *First-Class Phantom Types*. Technical Report CUCIS TR2003-1901, Cornell University.

[3] Thierry Coquand (1992): *Pattern Matching with Dependent Types*. In: *Proc. Workshop on Types for Proofs and Programs*, pp. 71–83.

[4] Joshua Dunfield (2007): *A Unified System of Type Refinements*. Ph.D. thesis, Carnegie Mellon University. CMU-CS-07-129.

[5] Jacques Garrigue (1998): *Programming with Polymorphic Variants*. In: *ML Workshop*, Baltimore.

[6] Jacques Garrigue & Jacques Le Normand (2011): *Adding GADTs to OCaml: the direct approach*. In: *Workshop on ML*, Tokyo.

[7] Georgios Karachalias, Tom Schrijvers, Dimitrios Vytiniotis & Simon Peyton Jones (2015): *GADTs meet their match*. In: *Proc. International Conference on Functional Programming*, pp. 424–436, doi:10.1145/2784731.2784748.

[8] Neelakantan R. Krishnaswami (2009): *Focusing on pattern matching*. In: *Proc. ACM Symposium on Principles of Programming Languages*, pp. 366–378, doi:10.1145/1480881.1480927.

[9] Fabrice Le Fessant & Luc Maranget (2001): *Optimizing Pattern Matching*. In: *Proc. International Conference on Functional Programming*, pp. 26–37, doi:10.1145/507635.507641.

[10] Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy & Jérôme Vouillon (2012): *The OCaml system release 4.00, Documentation and user's manual*. Projet Gallium, INRIA.

[11] Luc Maranget (2007): *Warnings for pattern matching*. *Journal of Functional Programming* 17(3), pp. 387–421, doi:10.1017/S0956796807006223.

[12] Conor McBride (1999): *Dependently Typed Functional Programs and their Proofs*. Doctor of philosophy, University of Edinburgh.

[13] Ulf Norell (2007): *Towards a practical programming language based on dependent type theory*. Ph.D. thesis, Chalmers University of Technology and Göteborg University.

[14] Ulf Norell (2009): *Dependently Typed Programming in Agda*. In: *Advanced Functional Programming 2008*, *Springer LNCS* 5832, pp. 230–266, doi:10.1007/978-3-642-04652-0_5.

[15] Nicolas Oury (2007): *Pattern Matching Coverage Checking with Dependent Types Using Set Approximations*. In: *Proc. Workshop on Programming Languages Meets Program Verification*, pp. 47–56, doi:10.1145/1292597.1292606.

[16] Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich & Geoffrey Washburn (2006): *Simple unification-based type inference for GADTs*. In: *Proc. International Conference on Functional Programming*, pp. 50–61, doi:10.1145/1159803.1159811.

[17] Carsten Schürmann & Frank Pfenning (2003): *A Coverage Checking Algorithm for LF*. In: *Proc. Theorem Proving in Higher Order Logics, Springer LNCS* 2758, pp. 120–135, doi:10.1007/10930755_8.

[18] OCaml bug tracker (2014): *GADT exhaustiveness check incompleteness*. OCaml problem report #6437. http://caml.inria.fr/mantis/view.php?id=6437.

[19] Hongwei Xi (1999): *Dead Code Elimination through Dependent Types*. In: *Proc. First International Workshop on Practical Aspects of Declarative Languages*, pp. 228–242, doi:10.1007/3-540-49201-1_16.

[20] Hongwei Xi, Chiyan Chen & Gang Chen (2003): *Guarded Recursive Datatype Constructors*. In: *Proc. ACM Symposium on Principles of Programming Languages*, ACM Press, pp. 224–235, doi:10.1145/604131.604150.