

# Real world GADTs

[Generalized algebraic data types](#) (GADTs) are an obtuse feature in languages like OCaml. They're like [algebraic data types](#), but they somehow "[make it possible to describe a richer relation between your data constructors and the types they inhabit](#)".

This is certainly true, but not a very helpful unless you are a type theorist. Humans learn by starting with the concrete and then generalizing to the abstract (*pun certainly intended*).

Most examples online demonstrate a fairly abstract example that you would rarely write in day-to-day code (for example, [evaluating an AST](#)). I don't personally write AST evaluators that often. But there are actual real uses for GADTs that you might find useful.

In this article, I go over *concrete* use-cases for GADTs, and then hope that you can start to generalize to the abstract essence of what a GADT is. Even if you can't, you'll be equipped with some patterns that you can apply, even if you don't understand the essence. (And I'll be frank: even I, the author of this article, do not yet have a solid intuition for GADTs.)

## Type maps

You can use GADTs as a way of mapping an input type to an output type.

TypeScript has similar (but much more intuitive) notions of this, via its [mapped types](#) and [conditional types](#). For example, you can define a type that takes one input type and produces an output type depending on what that input type was:

```
type SerializerOf<T> =  
  T extends string ? StringSerializer :  
  T extends number ? "number" :  
  T extends boolean ? "boolean" :  
  T extends undefined ? "undefined" :  
  T extends Function ? "function" :  
  "object";  
  
type T0 = SerializerOf<string>; // "string"  
type T1 = TypeName<"a">; // "string"  
type T2 = TypeName<true>; // "boolean"  
type T3 = TypeName<() => void>; // "function"  
type T4 = TypeName<string[]>; // "object"
```

In this example, `TypeName` is basically a function that maps an input type to an output type.

You could also use *associated types* in other languages (such as Rust or Swift) to accomplish a type mapping, although the syntax is usually heavier-weight.

GADTs in OCaml will let us do a similar thing, but with a less intuitive syntax:

```
type _ serializer =  
| Serialize_string: string -> string_serializer serializer  
| String_to_int: string -> int foo  
| Int_to_int_list: int -> int list foo
```

```

let serialize
let process_foo: type a. a foo -> a = fun foo ->
  match foo with
  | String_to_int (s: string) ->
    let result: int = int_of_string s in
    result
  | Int_to_int_list (i: int) ->
    let result: int list = [ i; i; i ] in
    result

```

The type `foo` here is a GADT that takes one type parameter. GADTs usually have their type parameter named `_`. This is because when you need to use the type parameter in a variant case of the GADT, you do it in the case definition rather than in the type definition, as we'll see later.

The syntax looks kind of like the function type syntax. Here, it means that a `String_to_int` value can be constructed using a parameter of type `string`, and the result will be a value of type `int foo`. I recommend treating this as a rote-memorization sort of rule, rather than delving into the intuition behind it.

Sometimes, when you match against a parameter that's a GADT, you need to explicitly annotate the type of the function with a `type a.` declaration. I don't understand exactly when you need to do this. I think newer versions of the OCaml compiler will figure out the types for you more intelligently.

In this code sample, we should read any instance of `a foo` as the input type, and `a` as the corresponding output type. Then we see that our `process_foo` function takes in a value of the input type `a foo` and produces a value of the output type `a`. In this case, one branch of the match returns an `int` and the other returns an `int list`, which are completely different types!

When would you want to do this? One example is a type-safe RPC system:

```

type _ message =
  | Add_one: int -> int message
  | Print_string: string -> unit message

let rpc: type a. a message -> a = fun message ->
  (* Could e.g. make a network call here instead. *)
  match message with
  | Add_one i ->
    i + 1
  | Print_string s ->
    print_endline s

let () =
  let four = rpc (Add_one 3) in
  rpc (Print_string (string_of_int four)) (* prints 4 *)

```

The alternative would be to make separate `request` and `response` variant types, and then manually assert that the response value is valid for the request value whenever you do an RPC call.

## Heterogeneous containers

Let's say we want to store our GADT-based messages into a mutable message queue now:

```
let () =  
  let q: 'a message Queue.t = Queue.create () in  
  Queue.push (Add_one 3) q;  
  Queue.push (Print_string "foo") q
```

This actually won't compile:

File "prog.ml", line 16, characters 34-35:

Error: This expression has type int message Queue.t  
 but an expression was expected of type unit message Queue.t  
 Type int is not compatible with type unit

The OCaml compiler is trying very hard to find a type to fill in for 'a, and it eagerly assumes that the first possible 'a is the one that it should use (this behavior is ultimately due to mutable state and the [value restriction](#)).

Actually, the type parameters to our message type are kind of meaningless until we extract an element from the queue, and we want to tell OCaml to ignore them for inference purposes until then.

This can be done with an [existential type](#). An existential type describes some type which is known to exist, but which remains abstract to the caller. We want to tell OCaml here that “there is some type parameter associated with each individual message, but you shouldn't care about it right now”.

### Existential types

One way of representing existential types in OCaml is with opaque type definitions like type t in a module declaration. Another way is with a GADT:

```
type queue_element =  
  | Element: 'a message -> queue_element  
  
let () =  
  let q: queue_element Queue.t = Queue.create () in  
  Queue.push (Element (Add_one 3)) q;  
  Queue.push (Element (Print_string "foo")) q
```

Here, we're defining the queue\_element type with exactly one case. That case acts as an existential wrapper around the message type. So what about it makes it “existential”, versus a regular, non-existential GADT?

Let's look at a regular variant type that takes a polymorphic type variable:

```
type 'a my_list =  
  | Nil  
  | Cons of 'a * 'a my_list  
  
let () =
```

```
let _my_list: int my_list = Cons (1, Cons (2, Nil)) in
()
```

Here's the equivalent type, but written using GADT syntax:

```
type 'a my_list =
| Nil: 'a my_list
| Cons: 'a * 'a my_list -> 'a my_list

let () =
let _my_list: int my_list = Cons (1, Cons (2, Nil)) in
()
```

The important thing to notice is that an 'a appears on the right-hand side of the arrow in each case. (Nil has no left-hand side, but it's kind of like saying unit -> 'a my\_list).

Most polymorphic variant types do this.

Let's look at our existential wrapper again:

```
type queue_element =
| Element: 'a message -> queue_element
```

This variant case has a polymorphic type variable on the left-hand side of an arrow, but not on the right-hand side. Additionally, queue\_element doesn't take a type variable as a parameter. It consumes a polymorphic type and produces a monomorphic type, so in effect, it "hides" the polymorphic type variable.

### Consuming existential types

Now, when we extract values from the queue, we need to get the message type back. We do this by pattern-matching on the value of type queue\_element in the usual way:

```
type _ message =
| Add_one: int -> int message
| Print_string: string -> unit message

type queue_element =
| Element: 'a message -> queue_element

let rpc (q: queue_element Queue.t): unit =
  match Queue.pop q with
  | Element (Add_one i) ->
    Printf.printf "Added one to %d to produce %d\n" i (i + 1)
  | Element (Print_string s) ->
    print_endline s

let () =
  let q: queue_element Queue.t = Queue.create () in
  Queue.push (Element (Add_one 3)) q;
  Queue.push (Element (Print_string "foo")) q;
```

```
rpc q; (* Prints "Added one to 3 to produce 4")
rpc q (* Prints "foo" *)
```

## Polymorphic type maps

We've covered two cases of using GADTs: one with no polymorphic type variables in the cases with type maps, and one with an input polymorphic type variable, but no output polymorphic type variable. For completeness, we'll discuss a case with an input polymorphic type variable and an output polymorphic type variable.

We'll take the RPC example from before, but add a polymorphic RPC method. It's very simple: just add 'a' to the input and output type where appropriate. In this example, `Make_pair` is a polymorphic RPC method:

```
type _ message =
| Add_one: int -> int message
| Print_string: string -> unit message
| Make_pair: 'a -> ('a * 'a) message

let rpc: type a. a message -> a = fun message ->
(* Could e.g. make a network call here instead. *)
match message with
| Add_one i ->
  i + 1
| Print_string s ->
  print_endline s
| Make_pair x ->
  (x, x)

let () =
  let four = rpc (Add_one 3) in
  let (_four, also_four) = rpc (Make_pair four) in
  rpc (Print_string (string_of_int also_four)) (* prints 4 *)
```