10 GADTs

Generalized Algebraic Data Types, or GADTs for short, are an extension of the variants we saw in Chapter 7 (Variants). GADTs are more expressive than regular variants, which helps you create types that more precisely match the shape of the program you want to write. That can help you write code that's safer, more concise, and more efficient.

At the same time, GADTs are an advanced feature of OCaml, and their power comes at a distinct cost. GADTs are harder to use and less intuitive than ordinary variants, and it can sometimes be a bit of a puzzle to figure out how to use them effectively. All of which is to say that you should only use a GADT when it makes a big qualitative improvement to your design.

That said, for the right use-case, GADTs can be really transformative, and this chapter will walk through several examples that demonstrate the range of use-cases that GADTs support.

At their heart, GADTs provide two extra features above and beyond ordinary variants:

- They let the compiler learn more type information when you descend into a case of a pattern match.
- They make it easy to use *existential types*, which let you work with data of a specific but unknown type.

It's a little hard to understand these features without working through some examples, so we'll do that next.

10.1 A Little Language

One classic use-case for GADTs is for writing typed expression languages, similar to the boolean expression language described in Chapter 7.3 (Variants and Recursive Data Structures). In this section, we'll create a slightly richer language that lets us mix arithmetic and boolean expressions. This means that we have to deal with the possibility of ill-typed expressions, e.g., an expression that adds a `bool` and an `int`.

Let's first try to do this with an ordinary variant. We'll declare two types here: `value`, which represents a primitive value in the language (i.e., an `int` or a `bool`), and `expr`, which represents the full set of possible expressions.

```
open Base

type value =
  | Int of int
  | Bool of bool

type expr =
  | Value of value
  | Eq of expr * expr
  | Plus of expr * expr
  | If of expr * expr * expr
```

We can write a recursive evaluator for this type in a pretty straight-ahead style. First, we'll declare an exception that can be thrown when we hit an ill-typed expression, e.g., when encountering an expression that tries to add a bool and an int.

```
exception Ill_typed
```

With that in hand, we can write the evaluator itself.

```
# let rec eval expr =
    match expr with
    | Value v -> v
    | If (c, t, e) ->
      (match eval c with
       | Bool b -> if b then eval t else eval e
       | Int _ -> raise Ill_typed)
    | Eq (x, y) ->
      (match eval x, eval y with
       | Bool _, _ | _, Bool _ -> raise Ill_typed
       | Int f1, Int f2 -> Bool (f1 = f2))
    | Plus (x, y) ->
      (match eval x, eval y with
       | Bool _, _ | _, Bool _ -> raise Ill_typed
       | Int f1, Int f2 -> Int (f1 + f2));;
val eval : expr -> value = <fun>
```

This implementation is a bit ugly because it has a lot of dynamic checks to detect type errors. Indeed, it's entirely possible to create an ill-typed expression which will trip these checks.

```
# let i x = Value (Int x)
  and b x = Value (Bool x)
  and (+:) x y = Plus (x,y);;
val i : int -> expr = <fun>
val b : bool -> expr = <fun>
val ( +: ) : expr -> expr -> expr = <fun>
# eval (i 3 +: b false);;
Exception: Ill_typed.
```

This possibility of ill-typed expressions doesn't just complicate the implementation: it's also a problem for users, since it's all too easy to create ill-typed expressions by mistake.

10.1.1 Making the Language Type-Safe

Let's consider what a type-safe version of this API might look like in the absence of GADTs. To even express the type constraints, we'll need expressions to have a type parameter to distinguish integer expressions from boolean expressions. Given such a parameter, the signature for such a language might look like this.

```
module type Typesafe_lang_sig = sig
  type 'a t

  (** functions for constructing expressions *)

  val int : int -> int t
  val bool : bool -> bool t
  val if_ : bool t -> 'a t -> 'a t -> 'a t
  val eq : 'a t -> 'a t -> bool t
  val plus : int t -> int t -> int t

  (** Evaluation functions *)

  val int_eval : int t -> int
  val bool_eval : bool t -> bool
end
```

The functions `int_eval` and `bool_eval` deserve some explanation. You might expect there to be a single evaluation function, with this signature.

```
val eval : 'a t -> 'a
```

But as we'll see, we're not going to be able to implement that, at least, not without using GADTs. So for now, we're stuck with two different evaluators, one for each type of expression.

Now let's write an implementation that matches this signature.

```
module Typesafe_lang : Typesafe_lang_sig = struct
  type 'a t = expr

  let int x = Value (Int x)
  let bool x = Value (Bool x)
  let if_ c t e = If (c, t, e)
  let eq x y = Eq (x, y)
  let plus x y = Plus (x, y)

  let int_eval expr =
    match eval expr with
    | Int x -> x
    | Bool _ -> raise Ill_typed

  let bool_eval expr =
    match eval expr with
    | Bool x -> x
    | Int _ -> raise Ill_typed
end
```

As you can see, the ill-typed expression we had trouble with before can't be constructed, because it's rejected by OCaml's type-system.

```
# let expr = Typesafe_lang.(plus (int 3) (bool false));;
Line 1, characters 40-52:
Error: This expression has type bool t but an expression was expected
    of type
         int t
       Type bool is not compatible with type int
```

So, what happened here? How did we add the type-safety we wanted? The fundamental trick is to add what's called a *phantom type*. In this definition:

```
type 'a t = expr
```

the type parameter `'a` is the phantom type, since it doesn't show up in the body of the definition of `t`.

Because the type parameter is unused, it's free to take on any value. That means we can constrain the use of that type parameter arbitrarily in the signature, which is a freedom we use to add the type-safety rules that we wanted.

This all amounts to an improvement in terms of the API, but the implementation is if anything worse. We still have the same evaluator with all of its dynamic checking for type errors. But we've had to write yet more wrapper code to make this work.

Also, the phantom-type discipline is quite error prone. You might have missed the fact that the type on the `eq` function above is wrong!

```
# Typesafe_lang.eq;;
- : 'a Typesafe_lang.t -> 'a Typesafe_lang.t -> bool Typesafe_lang.t =
    <fun>
```

It looks like it's polymorphic over the type of expressions, but the evaluator only supports checking equality on integers. As a result, we can still construct an ill-typed expression, phantom-types notwithstanding.

```
# let expr = Typesafe_lang.(eq (bool true) (bool false));;
val expr : bool Typesafe_lang.t = <abstr>
# Typesafe_lang.bool_eval expr;;
Exception: Ill_typed.
```

This highlights why we still need the dynamic checks in the implementation: the types within the implementation don't necessarily rule out ill-typed expressions. The same fact explains why we needed two different `eval` functions: the implementation of eval doesn't have any type-level guarantee of when it's handling a bool expression versus an int expression, so it can't safely give results where the type of the result varies based on the result of the expression.

10.1.2 Trying to Do Better with Ordinary Variants

To see why we need GADTs, let's see how far we can get without them. In particular, let's see what happens when we try to encode the typing rules we want for our DSL directly into the definition of the expression type. We'll do that by putting an ordinary type parameter on our `expr` and `value` types, in order to represent the type of an expression or value.

```
type 'a value =
  | Int of 'a
  | Bool of 'a

type 'a expr =
  | Value of 'a value
  | Eq of 'a expr * 'a expr
  | Plus of 'a expr * 'a expr
  | If of bool expr * 'a expr * 'a expr
```

This looks promising at first, but it doesn't quite do what we want. Let's experiment a little.

```
# let i x = Value (Int x)
  and b x = Value (Bool x)
  and (+:) x y = Plus (x,y);;
val i : 'a -> 'a expr = <fun>
val b : 'a -> 'a expr = <fun>
val ( +: ) : 'a expr -> 'a expr -> 'a expr = <fun>
# i 3;;
- : int expr = Value (Int 3)
# b false;;
- : bool expr = Value (Bool false)
# i 3 +: i 4;;
- : int expr = Plus (Value (Int 3), Value (Int 4))
```

So far so good. But if you think about it for a minute, you'll realize this doesn't actually do what we want. For one thing, the type of the outer expression is always just equal to the type of the inner expression, which means that some things that should type-check don't.

```
# If (Eq (i 3, i 4), i 0, i 1);;
Line 1, characters 9-12:
Error: This expression has type int expr
       but an expression was expected of type bool expr
       Type int is not compatible with type bool
```

Also, some things that shouldn't typecheck do.

```
# b 3;;
- : int expr = Value (Bool 3)
```

The problem here is that the way we want to use the type parameter isn't supported by ordinary variants. In particular, we want the type parameter to be populated in different ways in the different tags, and to depend in non-trivial ways on the types of the data associated with each tag. That's where GADTs can help.

10.1.3 GADTs to the Rescue

Now we're ready to write our first GADT. Here's a new version of our `value` and `expr` types that correctly encode our desired typing rules.

```
type _ value =
  | Int : int -> int value
  | Bool : bool -> bool value
```

```
type _ expr =
  | Value : 'a value -> 'a expr
  | Eq : int expr * int expr -> bool expr
  | Plus : int expr * int expr -> int expr
  | If : bool expr * 'a expr * 'a expr -> 'a expr
```

The syntax here requires some decoding. The colon to the right of each tag is what tells you that this is a GADT. To the right of the colon, you'll see what looks like an ordinary, single-argument function type, and you can almost think of it that way; specifically, as the type signature for that particular tag, viewed as a type constructor. The left-hand side of the arrow states the types of the arguments to the constructor, and the right-hand side determines the type of the constructed value.

In the definition of each tag in a GADT, the right-hand side of the arrow is an instance of the type of the overall GADT, with independent choices for the type parameter in each case. Importantly, the type parameter can depend both on the tag and on the type of the arguments. `Eq` is an example where the type parameter is determined entirely by the tag: it always corresponds to a `bool expr`. `If` is an example where the type parameter depends on the arguments to the tag, in particular the type parameter of the `If` is the type parameter of the then and else clauses.

Let's try some examples.

```
# let i x = Value (Int x)
  and b x = Value (Bool x)
  and (+:) x y = Plus (x,y);;
val i : int -> int expr = <fun>
val b : bool -> bool expr = <fun>
val ( +: ) : int expr -> int expr -> int expr = <fun>
# i 3;;
- : int expr = Value (Int 3)
# b 3;;
Line 1, characters 3-4:
Error: This expression has type int but an expression was expected of
     type
         bool
# i 3 +: i 6;;
- : int expr = Plus (Value (Int 3), Value (Int 6))
# i 3 +: b false;;
Line 1, characters 8-15:
Error: This expression has type bool expr
       but an expression was expected of type int expr
       Type bool is not compatible with type int
```

What we see here is that the type-safety rules we previously enforced with signature-level restrictions on phantom types are now directly encoded in the definition of the expression type.

These type-safety rules apply not just when constructing an expression, but also when deconstructing one, which means we can write a simpler and more concise evaluator that doesn't need any type-safety checks.

```
# let eval_value : type a. a value -> a = function
    | Int x -> x
    | Bool x -> x;;
val eval_value : 'a value -> 'a = <fun>
# let rec eval : type a. a expr -> a = function
    | Value v -> eval_value v
    | If (c, t, e) -> if eval c then eval t else eval e
    | Eq (x, y) -> eval x = eval y
    | Plus (x, y) -> eval x + eval y;;
val eval : 'a expr -> 'a = <fun>
```

Note that we now have a single polymorphic eval function, as opposed to the two type-specific evaluators we needed when using phantom types.

10.1.4 GADTs, Locally Abstract Types, and Polymorphic Recursion

The above example lets us see one of the downsides of GADTs, which is that code using them needs extra type annotations. Look at what happens if we write the definition of `value` without the annotation.

```
# let eval_value = function
    | Int x -> x
    | Bool x -> x;;
Line 3, characters 7-13:
Error: This pattern matches values of type bool value
       but a pattern was expected which matches values of type int
    value
       Type bool is not compatible with type int
```

The issue here is that OCaml by default isn't willing to instantiate ordinary type variables in different ways in the body of the same function, which is what is required here. We can fix that by adding a *locally abstract type*, which doesn't have that restriction.

```
# let eval_value (type a) (v : a value) : a =
    match v with
    | Int x -> x
    | Bool x -> x;;
val eval_value : 'a value -> 'a = <fun>
```

This isn't the same annotation we wrote earlier, and indeed, if we try this approach with `eval`, we'll see that it doesn't work.

```
# let rec eval (type a) (e : a expr) : a =
    match e with
    | Value v -> eval_value v
    | If (c, t, e) -> if eval c then eval t else eval e
    | Eq (x, y) -> eval x = eval y
    | Plus (x, y) -> eval x + eval y;;
Line 4, characters 43-44:
Error: This expression has type a expr but an expression was expected
    of type
         bool expr
       The type constructor a would escape its scope
```

This is a pretty unhelpful error message, but the basic problem is that `eval` is recursive, and inference of GADTs doesn't play well with recursive calls.

More specifically, the issue is that the type-checker is trying to merge the locally abstract type a into the type of the recursive function eval, and merging it into the outer scope within which eval is defined is the way in which a is escaping its scope.

We can fix this by explicitly marking eval as polymorphic, which OCaml has a handy type annotation for.

```
# let rec eval : 'a. 'a expr -> 'a =
    fun (type a) (x : a expr) ->
     match x with
      | Value v -> eval_value v
      | If (c, t, e) -> if eval c then eval t else eval e
      | Eq (x, y) -> eval x = eval y
      | Plus (x, y) -> eval x + eval y;;
val eval : 'a expr -> 'a = <fun>
```

This works because by marking eval as polymorphic, the type of eval isn't specialized to a, and so a doesn't escape its scope.

It's also helpful here because eval itself is an example of *polymorphic recursion*, which is to say that eval needs to call itself at multiple different types. This comes up, for example, with If, since the If itself must be of type bool, but the type of the then and else clauses could be of type int. This means that when evaluating If, we'll dispatch eval at a different type than it was called on.

As such, eval needs to see itself as polymorphic. This kind of polymorphism is basically impossible to infer automatically, which is a second reason we need to annotate eval's polymorphism explicitly.

The above syntax is a bit verbose, so OCaml has syntactic sugar to combine the polymorphism annotation and the creation of the locally abstract types:

```
# let rec eval : type a. a expr -> a = function
    | Value v -> eval_value v
    | If (c, t, e) -> if eval c then eval t else eval e
    | Eq (x, y) -> eval x = eval y
    | Plus (x, y) -> eval x + eval y;;
val eval : 'a expr -> 'a = <fun>
```

This type of annotation is the right one to pick when you write any recursive function that makes use of GADTs.

10.2 When Are GADTs Useful?

The typed language we showed above is a perfectly reasonable example, but GADTs are useful for a lot more than designing little languages. In this section, we'll try to give you a broader sampling of the kinds of things you can do with GADTs.

10.2.1 Varying Your Return Type

Sometimes, you want to write a single function that can effectively have different types in different circumstances. In some sense, this is totally ordinary. After all, OCaml's polymorphism means that values can take on different types in different contexts. `List.find` is a fine example. The signature indicates that the type of the result varies with the type of the input list.

```
# List.find;;
- : 'a list -> f:('a -> bool) -> 'a option = <fun>
```

And of course you can use `List.find` to produce values of different types.

```
# List.find ~f:(fun x -> x > 3) [1;3;5;2];;
- : int option = Some 5
# List.find ~f:(Char.is_uppercase) ['a';'B';'C'];;
- : char option = Some B
```

But this approach is limited to simple dependencies between types that correspond to how data flows through your code. Sometimes you want types to vary in a more flexible way.

To make this concrete, let's say we wanted to create a version of `find` that is configurable in terms of how it handles the case of not finding an item. There are three different behaviors you might want:

- Throw an exception.
- Return `None`.
- Return a default value.

Let's try to write a function that exhibits these behaviors without using GADTs. First, we'll create a variant type that represents the three possible behaviors.

```
module If_not_found = struct
  type 'a t =
    | Raise
    | Return_none
    | Default_to of 'a
end
```

Now we can write `flexible_find`, which takes an `If_not_found.t` as a parameter and varies its behavior accordingly.

```
# let rec flexible_find list ~f (if_not_found : _ If_not_found.t) =
    match list with
    | hd :: tl ->
      if f hd then Some hd else flexible_find ~f tl if_not_found
    | [] ->
      (match if_not_found with
      | Raise -> failwith "Element not found"
      | Return_none -> None
      | Default_to x -> Some x);;
val flexible_find :
  'a list -> f:('a -> bool) -> 'a If_not_found.t -> 'a option = <fun>
```

Here are some examples of the above function in action:

```
# flexible_find ~f:(fun x -> x > 10) [1;2;5] Return_none;;
- : int option = None
# flexible_find ~f:(fun x -> x > 10) [1;2;5] (Default_to 10);;
- : int option = Some 10
```

```
# flexible_find ~f:(fun x -> x > 10) [1;2;5] Raise;;
Exception: (Failure "Element not found")
# flexible_find ~f:(fun x -> x > 10) [1;2;20] Raise;;
- : int option = Some 20
```

This mostly does what we want, but the problem is that `flexible_find` always returns an option, even when it's passed `Raise` or `Default_to`, which guarantees that the `None` case is never used.

To eliminate the unnecessary option in the `Raise` and `Default_to` cases, we're going to turn `If_not_found.t` into a GADT. In particular, we'll mint it as a GADT with two type parameters: one for the type of the list element, and one for the return type of the function.

```
module If_not_found = struct
  type (_, _) t =
    | Raise : ('a, 'a) t
    | Return_none : ('a, 'a option) t
    | Default_to : 'a -> ('a, 'a) t
end
```

As you can see, `Raise` and `Default_to` both have the same element type and return type, but `Return_none` provides an optional return value.

Here's a definition of `flexible_find` that takes advantage of this GADT.

```
# let rec flexible_find
   : type a b. f:(a -> bool) -> a list -> (a, b) If_not_found.t -> b =
   fun ~f list if_not_found ->
    match list with
    | [] ->
      (match if_not_found with
      | Raise -> failwith "No matching item found"
      | Return_none -> None
      | Default_to x -> x)
    | hd :: tl ->
      if f hd
      then (
        match if_not_found with
        | Raise -> hd
        | Return_none -> Some hd
        | Default_to _ -> hd)
      else flexible_find ~f tl if_not_found;;
val flexible_find :
  f:('a -> bool) -> 'a list -> ('a, 'b) If_not_found.t -> 'b = <fun>
```

As you can see from the signature of `flexible_find`, the return value now depends on the type of `If_not_found.t`, which means it can depend on the particular variant of `If_not_found.t` that's in use. As a result, `flexible_find` only returns an option when it needs to.

```
# flexible_find ~f:(fun x -> x > 10) [1;2;5] Return_none;;
- : int option = Base.Option.None
# flexible_find ~f:(fun x -> x > 10) [1;2;5] (Default_to 10);;
- : int = 10
# flexible_find ~f:(fun x -> x > 10) [1;2;5] Raise;;
Exception: (Failure "No matching item found")
```

```
# flexible_find ~f:(fun x -> x > 10) [1;2;20] Raise;;
- : int = 20
```

10.2.2 Capturing the Unknown

Code that works with unknown types is routine in OCaml, and comes up in the simplest of examples:

```
# let tuple x y = (x,y);;
val tuple : 'a -> 'b -> 'a * 'b = <fun>
```

The type variables `'a` and `'b` indicate that there are two unknown types here, and these type variables are *universally quantified*. Which is to say, the type of `tuple` is: *for all* types `a` and `b`, `a -> b -> a * b`.

And indeed, we can restrict the type of `tuple` to any `'a` and `'b` we want.

```
# (tuple : int -> float -> int * float);;
- : int -> float -> int * float = <fun>
# (tuple : string -> string * string -> string * (string * string));;
- : string -> string * string -> string * (string * string) = <fun>
```

Sometimes, however, we want type variables that are *existentially quantified*, meaning that instead of being compatible with all types, the type represents a particular but unknown type.

GADTs provide one natural way of encoding such type variables. Here's a simple example.

```
type stringable =
  Stringable : { value: 'a; to_string: 'a -> string } -> stringable
```

This type packs together a value of some arbitrary type, along with a function for converting values of that type to strings.

We can tell that `'a` is existentially quantified because it shows up on the left-hand side of the arrow but not on the right, so the `'a` that shows up internally doesn't appear in a type parameter for `stringable` itself. Essentially, the existentially quantified type is bound within the definition of `stringable`.

The following function can print an arbitrary `stringable`:

```
# let print_stringable (Stringable s) =
    Stdio.print_endline (s.to_string s.value);;
val print_stringable : stringable -> unit = <fun>
```

We can use `print_stringable` on a collection of `stringables` of different underlying types.

```
# let stringables =
    (let s value to_string = Stringable { to_string; value } in
      [ s 100 Int.to_string
      ; s 12.3 Float.to_string
      ; s "foo" Fn.id
      ]);;
val stringables : stringable list =
  [Stringable {value = <poly>; to_string = <fun>};
```

```
   Stringable {value = <poly>; to_string = <fun>};
   Stringable {value = <poly>; to_string = <fun>}]
# List.iter ~f:print_stringable stringables;;
100
12.3
foo
- : unit = ()
```

The thing that lets this all work is that the type of the underlying object is existentially bound within the type `stringable`. As such, the type of the underlying values can't escape the scope of `stringable`, and any function that tries to return such a value won't type-check.

```
# let get_value (Stringable s) = s.value;;
Line 1, characters 32-39:
Error: This expression has type $Stringable_'a
       but an expression was expected of type 'a
       The type constructor $Stringable_'a would escape its scope
```

It's worth spending a moment to decode this error message, and the meaning of the type variable `$Stringable_'a` in particular. You can think of this variable as having three parts:

- The `$` marks the variable as an existential.
- `Stringable` is the name of the GADT tag that this variable came from.
- `'a` is the name of the type variable from inside that tag.

10.2.3 Abstracting Computational Machines

A common idiom in OCaml is to combine small components into larger computational machines, using a collection of component-combining functions, or *combinators*.

GADTs can be helpful for writing such combinators. To see how, let's consider an example: *pipelines*. Here, a pipeline is a sequence of steps where each step consumes the output of the previous step, potentially does some side effects, and returns a value to be passed to the next step. This is analogous to a shell pipeline, and is useful for all sorts of system automation tasks.

But, can't we write pipelines already? After all, OCaml comes with a perfectly serviceable pipeline operator:

```
# open Core;;
# let sum_file_sizes () =
    Sys.ls_dir "."
    |> List.filter ~f:Sys.is_file_exn
    |> List.map ~f:(fun file_name -> (Unix.lstat file_name).st_size)
    |> List.sum (module Int) ~f:Int64.to_int_exn;;
val sum_file_sizes : unit -> int = <fun>
```

This works well enough, but the advantage of a custom pipeline type is that it lets you build extra services beyond basic execution of the pipeline, e.g.:

- Profiling, so that when you run a pipeline, you get a report of how long each step of the pipeline took.

- Control over execution, like allowing users to pause the pipeline mid-execution, and restart it later.
- Custom error handling, so, for example, you could build a pipeline that kept track of where it failed, and offered the possibility of restarting it.

The type signature of such a pipeline type might look something like this:

```
module type Pipeline = sig
  type ('input,'output) t

  val ( @> ) : ('a -> 'b) -> ('b,'c) t -> ('a,'c) t
  val empty : ('a,'a) t
end
```

Here, the type `('a,'b) t` represents a pipeline that consumes values of type `'a` and emits values of type `'b`. The operator `@>` lets you add a step to a pipeline by providing a function to prepend on to an existing pipeline, and `empty` gives you an empty pipeline, which can be used to seed the pipeline.

The following shows how we could use this API for building a pipeline like our earlier example using `|>`. Here, we're using a *functor*, which we'll see in more detail in Chapter 11 (Functors), as a way to write code using the pipeline API before we've implemented it.

```
# module Example_pipeline (Pipeline : Pipeline) = struct
    open Pipeline
    let sum_file_sizes =
      (fun () -> Sys.ls_dir ".")
      @> List.filter ~f:Sys.is_file_exn
      @> List.map ~f:(fun file_name -> (Unix.lstat file_name).st_size)
      @> List.sum (module Int) ~f:Int64.to_int_exn
      @> empty
  end;;
module Example_pipeline :
  functor (Pipeline : Pipeline) ->
    sig val sum_file_sizes : (unit, int) Pipeline.t end
```

If all we want is a pipeline capable of a no-frills execution, we can define our pipeline itself as a simple function, the `@>` operator as function composition. Then executing the pipeline is just function application.

```
module Basic_pipeline : sig
   include Pipeline
   val exec : ('a,'b) t -> 'a -> 'b
 end= struct
  type ('input, 'output) t = 'input -> 'output

  let empty = Fn.id

  let ( @> ) f t input =
    t (f input)

  let exec t input = t input
end
```

But this way of implementing a pipeline doesn't give us any of the extra services we discussed. All we're really doing is step-by-step building up the same kind of function that we could have gotten using the |> operator.

We could get a more powerful pipeline by simply enhancing the pipeline type, providing it with extra runtime structures to track profiles, or handle exceptions, or provide whatever else is needed for the particular use-case. But this approach is awkward, since it requires us to pre-commit to whatever services we're going to support, and to embed all of them in our pipeline representation.

GADTs provide a simpler approach. Instead of concretely building a machine for executing a pipeline, we can use GADTs to abstractly represent the pipeline we want, and then build the functionality we want on top of that representation.

Here's what such a representation might look like.

```
type (_, _) pipeline =
  | Step : ('a -> 'b) * ('b, 'c) pipeline -> ('a, 'c) pipeline
  | Empty : ('a, 'a) pipeline
```

The tags here represent the two building blocks of a pipeline: Step corresponds to the @> operator, and Empty corresponds to the empty pipeline, as you can see below.

```
# let ( @> ) f pipeline = Step (f,pipeline);;
val ( @> ) : ('a -> 'b) -> ('b, 'c) pipeline -> ('a, 'c) pipeline = <fun>
# let empty = Empty;;
val empty : ('a, 'a) pipeline = Empty
```

With that in hand, we can do a no-frills pipeline execution easily enough.

```
# let rec exec : type a b. (a, b) pipeline -> a -> b =
   fun pipeline input ->
    match pipeline with
    | Empty -> input
    | Step (f, tail) -> exec tail (f input);;
val exec : ('a, 'b) pipeline -> 'a -> 'b = <fun>
```

But we can also do more interesting things. For example, here's a function that executes a pipeline and produces a profile showing how long each step of a pipeline took.

```
# let exec_with_profile pipeline input =
    let rec loop
        : type a b.
          (a, b) pipeline -> a -> Time_ns.Span.t list -> b *
    Time_ns.Span.t list
      =
     fun pipeline input rev_profile ->
      match pipeline with
      | Empty -> input, rev_profile
      | Step (f, tail) ->
        let start = Time_ns.now () in
        let output = f input in
        let elapsed = Time_ns.diff (Time_ns.now ()) start in
        loop tail output (elapsed :: rev_profile)
    in
    let output, rev_profile = loop pipeline input [] in
    output, List.rev rev_profile;;
```

```
val exec_with_profile : ('a, 'b) pipeline -> 'a -> 'b * Time_ns.Span.t
    list =
  <fun>
```

The more abstract GADT approach for creating a little combinator library like this has several advantages over having combinators that build a more concrete computational machine:

- The core types are simpler, since they are typically built out of GADT tags that are just reflections of the types of the base combinators.
- The design is more modular, since your core types don't need to contemplate every possible use you want to make of them.
- The code tends to be more efficient, since the more concrete approach typically involves allocating closures to wrap up the necessary functionality, and closures are more heavyweight than GADT tags.

10.2.4 Narrowing the Possibilities

Another use-case for GADTs is to narrow the set of possible states for a given data-type in different circumstances.

One context where this can be useful is when managing complex application state, where the available data changes over time. Let's consider a simple example, where we're writing code to handle a logon request from a user, and we want to check if the user in question is authorized to logon.

We'll assume that the user logging in is authenticated as a particular name, but that in order to authenticate, we need to do two things: to translate that user-name into a numeric user-id, and to fetch permissions for the service in question; once we have both, we can check if the user-id is permitted to log on.

Without GADTs, we might model the state of a single logon request as follows.

```
type logon_request =
  { user_name : User_name.t
  ; user_id : User_id.t option
  ; permissions : Permissions.t option
  }
```

Here, `User_name.t` represents a textual name, `User_id.t` represents an integer identifier associated with a user, and a `Permissions.t` lets you determine which `User_id.t`'s are authorized to log in.

Here's how we might write a function for testing whether a given request is authorized.

```
# let authorized request =
    match request.user_id, request.permissions with
    | None, _ | _, None ->
      Error "Can't check authorization: data incomplete"
    | Some user_id, Some permissions ->
      Ok (Permissions.check permissions user_id);;
val authorized : logon_request -> (bool, string) result = <fun>
```

The intent is to only call this function once the data is complete, i.e., when the `user_id` and `permissions` fields have been filled in, which is why it errors out if the data is incomplete.

The code above works just fine for a simple case like this. But in a real system, your code can get more complicated in multiple ways, e.g.,

- more fields to manage, including more optional fields,
- more operations that depend on these optional fields,
- multiple requests to be handled in parallel, each of which might be in a different state of completion.

As this kind of complexity creeps in, it can be useful to be able to track the state of a given request at the type level, and to use that to narrow the set of states a given request can be in, thereby removing some extra case analysis and error handling, which can reduce the complexity of the code and remove opportunities for mistakes.

One way of doing this is to mint different types to represent different states of the request, e.g., one type for an incomplete request where various fields are optional, and a different type where all of the data is mandatory.

While this works, it can be awkward and verbose. With GADTs, we can track the state of the request in a type parameter, and have that parameter be used to narrow the set of available cases, without duplicating the type.

A Completion-Sensitive Option Type

We'll start by creating an option type that is sensitive to whether our request is in a complete or incomplete state. To do that, we'll mint types to represent the states of being complete and incomplete.

```
type incomplete = Incomplete
type complete = Complete
```

The definition of the types doesn't really matter, since we're never instantiating these types, just using them as markers of different states. All that matters is that the types are distinct.

Now we can mint a completeness-sensitive option type. Note the two type variables: the first indicates the type of the contents of the option, and the second indicates whether this is being used in an incomplete state.

```
type (_, _) coption =
  | Absent : (_, incomplete) coption
  | Present : 'a -> ('a, _) coption
```

We use `Absent` and `Present` rather than `Some` or `None` to make the code less confusing when both `option` and `coption` are used together.

You might notice that we haven't used `complete` here explicitly. Instead, what we've done is to ensure that only an `incomplete coption` can be `Absent`. Accordingly, a `coption` that's `complete` (and therefore not `incomplete`) can only be `Present`.

This is easier to understand with some examples. Consider the following function for getting the value out of a `coption`, returning a default value if `Absent` is found.

```
# let get ~default o =
     match o with
     | Present x -> x
     | Absent -> default;;
val get : default:'a -> ('a, incomplete) coption -> 'a = <fun>
```

Note that the incomplete type was inferred here. If we annotate the coption as complete, the code no longer compiles.

```
# let get ~default (o : (_,complete) coption) =
    match o with
    | Absent -> default
    | Present x -> x;;
Line 3, characters 7-13:
Error: This pattern matches values of type ('a, incomplete) coption
       but a pattern was expected which matches values of type
         ('a, complete) coption
       Type incomplete is not compatible with type complete
```

We can make this compile by deleting the Absent branch (and the now useless default argument).

```
# let get (o : (_,complete) coption) =
    match o with
    | Present x -> x;;
val get : ('a, complete) coption -> 'a = <fun>
```

We could write this more simply as:

```
# let get (Present x : (_,complete) coption) = x;;
val get : ('a, complete) coption -> 'a = <fun>
```

As we can see, when the coption is known to be complete, the pattern matching is narrowed to just the Present case.

A Completion-Sensitive Request Type

We can use coption to define a completion-sensitive version of logon_request.

```
type 'c logon_request =
  { user_name : User_name.t
  ; user_id : (User_id.t, 'c) coption
  ; permissions : (Permissions.t, 'c) coption
  }
```

There's a single type parameter for the logon_request that marks whether it's complete, at which point, both the user_id and permissions fields will be complete as well.

As before, it's easy to fill in the user_id and permissions fields.

```
# let set_user_id request x = { request with user_id = Present x };;
val set_user_id : 'a logon_request -> User_id.t -> 'a logon_request =
    <fun>
# let set_permissions request x = { request with permissions =
    Present x };;
val set_permissions : 'a logon_request -> Permissions.t -> 'a
    logon_request =
  <fun>
```

Note that filling in the fields doesn't automatically mark a request as `complete`. To do that, we need to explicitly test for completeness, and then construct a version of the record with just the completed fields filled in.

```
# let check_completeness request =
    match request.user_id, request.permissions with
    | Absent, _ | _, Absent -> None
    | (Present _ as user_id), (Present _ as permissions) ->
      Some { request with user_id; permissions };;
val check_completeness : incomplete logon_request -> 'a logon_request
    option =
  <fun>
```

The result is polymorphic, meaning it can return a logon request of any kind, which includes the possibility of returning a complete request. In practice, the function type is easier to understand if we constrain the return value to explicitly return a `complete` request.

```
# let check_completeness request : complete logon_request option =
    match request.user_id, request.permissions with
    | Absent, _ | _, Absent -> None
    | (Present _ as user_id), (Present _ as permissions) ->
      Some { request with user_id; permissions };;
val check_completeness :
  incomplete logon_request -> complete logon_request option = <fun>
```

Finally, we can write an authorization checker that works unconditionally on a complete login request.

```
# let authorized (request : complete logon_request) =
    let { user_id = Present user_id; permissions = Present
    permissions; _ } = request in
    Permissions.check permissions user_id;;
val authorized : complete logon_request -> bool = <fun>
```

After all that work, the result may seem a bit underwhelming, and indeed, most of the time, this kind of narrowing isn't worth the complexity of setting it up. But for a sufficiently complex state machine, cutting down on the possibilities that your code needs to contemplate can make a big difference to the comprehensibility and correctness of the result.

Type Distinctness and Abstraction

In the example in this section, we used two types, `complete` and `incomplete` to mark different states, and we defined those types so as to be in some sense obviously different.

```
type incomplete = Incomplete
type complete = Complete
```

This isn't strictly necessary. Here's another way of defining these types that makes them less obviously distinct.

```
type incomplete = Z
type complete = Z
```

OCaml's variant types are nominal, so `complete` and `incomplete` are distinct types, despite having variants of the same name, as you can see when we try to put instances of each type in the same list.

```
# let i = (Z : incomplete) and c = (Z : complete);;
val i : incomplete = Z
val c : complete = Z
# [i; c];;
Line 1, characters 5-6:
Error: This expression has type complete
       but an expression was expected of type incomplete
```

As a result, we can narrow a pattern match using these types as indices, much as we did earlier. First, we set up the `coption` type:

```
type ('a, _) coption =
  | Absent : (_, incomplete) coption
  | Present : 'a -> ('a, _) coption
```

Then, we write a function that requires the `coption` to be complete, and accordingly, need only contemplate the `Present` case.

```
# let assume_complete (coption : (_,complete) coption) =
    match coption with
    | Present x -> x;;
val assume_complete : ('a, complete) coption -> 'a = <fun>
```

An easy-to-miss issue here is that the way we expose these types through an interface can cause OCaml to lose track of the distinctness of the types in question. Consider this version, where we entirely hide the definition of `complete` and `incomplete`.

```
module M : sig
  type incomplete
  type complete
end = struct
  type incomplete = Z
  type complete = Z
end
include M

type ('a, _) coption =
  | Absent : (_, incomplete) coption
  | Present : 'a -> ('a, _) coption
```

Now, the `assume_complete` function we wrote is no longer found to be exhaustive.

```
# let assume_complete (coption : (_,complete) coption) =
    match coption with
    | Present x -> x;;
Lines 2-3, characters 5-21:
Warning 8 [partial-match]: this pattern-matching is not exhaustive.
Here is an example of a case that is not matched:
Absent
val assume_complete : ('a, complete) coption -> 'a = <fun>
```

That's because by leaving the types abstract, we've entirely hidden the underlying types, leaving the type system with no evidence that the types are distinct.

Let's see what happens if we expose the implementation of these types.

```
module M : sig
  type incomplete = Z
  type complete = Z
end = struct
  type incomplete = Z
  type complete = Z
end
include M

type ('a, _) coption =
  | Absent : (_, incomplete) coption
  | Present : 'a -> ('a, _) coption
```

But the result is still not exhaustive!

```
# let assume_complete (coption : (_,complete) coption) =
    match coption with
    | Present x -> x;;
Lines 2-3, characters 5-21:
Warning 8 [partial-match]: this pattern-matching is not exhaustive.
Here is an example of a case that is not matched:
Absent
val assume_complete : ('a, complete) coption -> 'a = <fun>
```

In order to be exhaustive, we need the types that are exposed to be definitively different, which would be the case if we defined them as variants with differently named tags, as we did originally.

The reason for this is that types that appear to be different in an interface may turn out to be the same in the implementation, as we can see below.

```
module M : sig
  type incomplete = Z
  type complete = Z
end = struct
  type incomplete = Z
  type complete = incomplete = Z
end
```

All of which is to say: when creating types to act as abstract markers for the type parameter of a GADT, you should choose definitions that make the distinctness of those types clear, and you should expose those definitions in your `mli`s.

Narrowing Without GADTs

Thus far, we've only seen narrowing in the context of GADTs, but OCaml can eliminate impossible cases from ordinary variants too. As with GADTs, to eliminate a case you need to demonstrate that the case in question is impossible at the type level.

One way to do this is via an *uninhabited type*, which is a type that has no associated values. You can declare such a value by creating a variant with no tags.

```
type nothing = |
```

This turns out to be useful enough that `Base` has a standard uninhabited type, `Nothing.t`.

So, how does an uninhabited type help? Well, consider the `Result.t` type, discussed in as described in Chapter 8.1.1 (Encoding Errors with Result). Normally, to match a `Result.t`, you need to handle both the `Ok` and `Error` cases.

```
# open Stdio;;
# let print_result (x : (int,string) Result.t) =
    match x with
    | Ok x -> printf "%d\n" x
    | Error x -> printf "ERROR: %s\n" x;;
val print_result : (int, string) result -> unit = <fun>
```

But if the `Error` case contains an uninhabitable type, well, that case can never be instantiated, and OCaml will tell you as much.

```
# let print_result (x : (int, Nothing.t) Result.t) =
    match x with
    | Ok x -> printf "%d\n" x
    | Error _ -> printf "ERROR\n";;
Line 4, characters 7-14:
Warning 56 [unreachable-case]: this match case is unreachable.
Consider replacing it with a refutation case '<pat> -> .'
val print_result : (int, Nothing.t) result -> unit = <fun>
```

We can follow the advice above, and add a so-called *refutation case*.

```
# let print_result (x : (int, Nothing.t) Result.t) =
    match x with
    | Ok x -> printf "%d\n" x
    | Error _ -> .;;
val print_result : (int, Nothing.t) result -> unit = <fun>
```

The period in the final case tells the compiler that we believe this case can never be reached, and OCaml will verify that it's true. In some simple cases, however, the compiler can automatically add the refutation case for you, so you don't need to write it out explicitly.

```
# let print_result (x : (int, Nothing.t) Result.t) =
    match x with
    | Ok x -> printf "%d\n" x;;
val print_result : (int, Nothing.t) result -> unit = <fun>
```

Narrowing with uninhabitable types can be useful when using a highly configurable library that supports multiple different modes of use, not all of which are necessarily needed for a given application. One example of this comes from `Async`'s RPC (remote procedure-call) library. Async RPCs support a particular flavor of interaction called a `State_rpc`. Such an RPC is parameterized by four types, for four different kinds of data:

- `query`, for the initial client request,
- `state`, for the initial snapshot returned by the server,
- `update`, for the sequence of updates to that snapshot, and
- `error`, for an error to terminate the stream.

Now, imagine you want to use a `State_rpc` in a context where you don't need to terminate the stream with a custom error. We could just instantiate the `State_rpc` using the type `unit` for the error type.

```
open Core
open Async
let rpc =
  Rpc.State_rpc.create
    ~name:"int-map"
    ~version:1
    ~bin_query:[%bin_type_class: unit]
    ~bin_state:[%bin_type_class: int Map.M(String).t]
    ~bin_update:[%bin_type_class: int Map.M(String).t]
    ~bin_error:[%bin_type_class: unit]
    ()
```

But with this approach, you still have to handle the error case when writing code to dispatch the RPC.

```
# let dispatch conn =
    match%bind Rpc.State_rpc.dispatch rpc conn () >>| ok_exn with
    | Ok (initial_state, updates, _) -> handle_state_changes
    initial_state updates
    | Error () -> failwith "this is not supposed to happen";;
val dispatch : Rpc.Connection.t -> unit Deferred.t = <fun>
```

An alternative approach is to use an uninhabited type for the error:

```
let rpc =
  Rpc.State_rpc.create
    ~name:"foo"
    ~version:1
    ~bin_query:[%bin_type_class: unit]
    ~bin_state:[%bin_type_class: int Map.M(String).t]
    ~bin_update:[%bin_type_class: int Map.M(String).t]
    ~bin_error:[%bin_type_class: Nothing.t]
    ()
```

Now, we've essentially banned the use of the `error` type, and as a result, our dispatch function needs only deal with the `Ok` case.

```
# let dispatch conn =
    match%bind Rpc.State_rpc.dispatch rpc conn () >>| ok_exn with
    | Ok (initial_state, updates, _) -> handle_state_changes
    initial_state updates;;
val dispatch : Rpc.Connection.t -> unit Deferred.t = <fun>
```

What's nice about this example is that it shows that narrowing can be applied to code that isn't designed with narrowing in mind.

10.3 Limitations of GADTs

Hopefully, we've demonstrated the utility of GADTs, while at the same time showing some of the attendant complexities. In this final section, we're going to highlight some remaining difficulties with using GADTs that you may run into, as well as how to work around them.

10.3.1 Or-Patterns

GADTs don't work well with or-patterns. Consider the following type that represents various ways we might use for obtaining some piece of data.

```
open Core
module Source_kind = struct
  type _ t =
    | Filename : string t
    | Host_and_port : Host_and_port.t t
    | Raw_data : string t
end
```

We can write a function that takes a `Source_kind.t` and the corresponding source, and prints it out.

```
# let source_to_sexp (type a) (kind : a Source_kind.t) (source : a) =
    match kind with
    | Filename -> String.sexp_of_t source
    | Host_and_port -> Host_and_port.sexp_of_t source
    | Raw_data -> String.sexp_of_t source;;
val source_to_sexp : 'a Source_kind.t -> 'a -> Sexp.t = <fun>
```

But, observing that the right-hand side of `Raw_data` and `Filename` are the same, you might try to merge those cases together with an or-pattern. Unfortunately, that doesn't work.

```
# let source_to_sexp (type a) (kind : a Source_kind.t) (source : a) =
    match kind with
    | Filename | Raw_data -> String.sexp_of_t source
    | Host_and_port -> Host_and_port.sexp_of_t source;;
Line 3, characters 47-53:
Error: This expression has type a but an expression was expected of
    type
         string
```

Or-patterns do sometimes work, but only when you don't make use of the type information that is discovered during the pattern match. Here's an example of a function that uses or-patterns successfully.

```
# let requires_io (type a) (kind : a Source_kind.t) =
    match kind with
    | Filename | Host_and_port -> true
    | Raw_data -> false;;
val requires_io : 'a Source_kind.t -> bool = <fun>
```

In any case, the lack of or-patterns is annoying, but it's not a big deal, since you can reduce the code duplication by pulling out most of the content of the duplicated right-hand sides into functions that can be called in each of the duplicated cases.

10.3.2 Deriving Serializers

As will be discussed in more detail in Chapter 21 (Data Serialization with S-Expressions), s-expressions are a convenient data format for representing structured data. Rather than write the serializers and deserializers by hand, we typically use ppx_sexp_value, which is a syntax extension which auto-generates these functions for a given type, based on that type's definition.

Here's an example:

```
# type position = { x: float; y: float } [@@deriving sexp];;
type position = { x : float; y : float; }
val position_of_sexp : Sexp.t -> position = <fun>
val sexp_of_position : position -> Sexp.t = <fun>
# sexp_of_position { x = 3.5; y = -2. };;
- : Sexp.t = ((x 3.5) (y -2))
# position_of_sexp (Sexp.of_string "((x 72) (y 1.2))");;
- : position = {x = 72.; y = 1.2}
```

While [@@deriving sexp] works with most types, it doesn't always work with GADTs.

```
# type _ number_kind =
    | Int : int number_kind
    | Float : float number_kind
  [@@deriving sexp];;
Lines 1-4, characters 1-20:
Error: This expression has type int number_kind
       but an expression was expected of type a__001_ number_kind
       Type int is not compatible with type a__001_
```

The error message is pretty awful, but if you stop and think about it, it's not too surprising that we ran into trouble here. What should the type of number_kind_of_sexp be anyway? When parsing "Int", the returned type would have to be int number_kind, and when parsing "Float", the type would have to be float number_kind. That kind of dependency between the value of an argument and the type of the returned value is just not expressible in OCaml's type system.

This argument doesn't stop us from serializing, and indeed, [@@deriving sexp_of], which only creates the serializer, works just fine.

```
# type _ number_kind =
   | Int : int number_kind
   | Float : float number_kind
  [@@deriving sexp_of];;
type _ number_kind = Int : int number_kind | Float : float number_kind
val sexp_of_number_kind :
  ('a__001_ -> Sexp.t) -> 'a__001_ number_kind -> Sexp.t = <fun>
# sexp_of_number_kind Int.sexp_of_t Int;;
- : Sexp.t = Int
```

It is possible to build a deserializer for number_kind, but it's tricky. First, we'll need a type that packs up a number_kind while hiding its type parameter. This is going to be the value we return from our parser.

```
type packed_number_kind = P : _ number_kind -> packed_number_kind
```

Next, we'll need to create a non-GADT version of our type, for which we'll derive a deserializer.

```
type simple_number_kind = Int | Float [@@deriving of_sexp]
```

Then, we write a function for converting from our non-GADT type to the packed variety.

```
# let simple_number_kind_to_packed_number_kind kind :
    packed_number_kind
    =
    match kind with
    | Int -> P Int
    | Float -> P Float;;
val simple_number_kind_to_packed_number_kind :
  simple_number_kind -> packed_number_kind = <fun>
```

Finally, we combine our generated sexp-converter with our conversion type to produce the full deserialization function.

```
# let number_kind_of_sexp sexp =
    simple_number_kind_of_sexp sexp
    |> simple_number_kind_to_packed_number_kind;;
val number_kind_of_sexp : Sexp.t -> packed_number_kind = <fun>
```

And here's that function in action.

```
# List.map ~f:number_kind_of_sexp
    [ Sexp.of_string "Float"; Sexp.of_string "Int" ];;
- : packed_number_kind list = [P Float; P Int]
```

While all of this is doable, it's definitely awkward, and requires some unpleasant code duplication.