7 Variants

Variant types are one of the most useful features of OCaml and also one of the most unusual. They let you represent data that may take on multiple different forms, where each form is marked by an explicit tag. As we'll see, when combined with pattern matching, variants give you a powerful way of representing complex data and of organizing the case-analysis on that information.

The basic syntax of a variant type declaration is as follows:

```
type <variant> =
  | <Tag> [ of <type> [* <type>]... ]
  | <Tag> [ of <type> [* <type>]... ]
  | ...
```

Each row essentially represents a case of the variant. Each case has an associated tag (also called a *constructor*, since you use it to construct a value) and may optionally have a sequence of fields, where each field has a specified type.

Let's consider a concrete example of how variants can be useful. Most UNIX-like operating systems support terminals as a fundamental, text-based user interface. Almost all of these terminals support a set of eight basic colors.

Those colors can be naturally represented as a variant. Each color is declared as a simple tag, with pipes used to separate the different cases. Note that variant tags must be capitalized.

```
open Base
open Stdio
type basic_color =
  | Black | Red | Green | Yellow | Blue | Magenta | Cyan | White
```

As we show below, the variant tags introduced by the definition of `basic_color` can be used for constructing values of that type.

```
# Cyan;;
- : basic_color = Cyan
# [Blue; Magenta; Red];;
- : basic_color list = [Blue; Magenta; Red]
```

The following function uses pattern matching to convert each of these to the corresponding integer code that is used for communicating these colors to the terminal.

```
# let basic_color_to_int = function
    | Black -> 0 | Red     -> 1 | Green -> 2 | Yellow -> 3
    | Blue  -> 4 | Magenta -> 5 | Cyan  -> 6 | White  -> 7;;
```

https://doi.org/10.1017/9781009129220.009 Published online by Cambridge University Press

```
val basic_color_to_int : basic_color -> int = <fun>
# List.map ~f:basic_color_to_int [Blue;Red];;
- : int list = [4; 1]
```

We know that the above function handles every color in `basic_color` because the compiler would have warned us if we'd missed one:

```
# let incomplete_color_to_int = function
    | Black -> 0 | Red -> 1 | White -> 7;;
Lines 1-2, characters 31-41:
Warning 8 [partial-match]: this pattern-matching is not exhaustive.
Here is an example of a case that is not matched:
(Green|Yellow|Blue|Magenta|Cyan)
val incomplete_color_to_int : basic_color -> int = <fun>
```

In any case, using the correct function, we can generate escape codes to change the color of a given string displayed in a terminal.

```
# let color_by_number number text =
  Printf.sprintf "\027[38;5;%dm%s\027[0m" number text;;
val color_by_number : int -> string -> string = <fun>
# let blue = color_by_number (basic_color_to_int Blue) "Blue";;
val blue : string = "\027[38;5;4mBlue\027[0m"
# printf "Hello %s World!\n" blue;;
Hello Blue World!
- : unit = ()
```

On most terminals, that word "Blue" will be rendered in blue.

In this example, the cases of the variant are simple tags with no associated data. This is substantively the same as the enumerations found in languages like C and Java. But as we'll see, variants can do considerably more than represent simple enumerations.

As it happens, an enumeration isn't enough to effectively describe the full set of colors that a modern terminal can display. Many terminals, including the venerable `xterm`, support 256 different colors, broken up into the following groups:

- The eight basic colors, in regular and bold versions
- A $6 \times 6 \times 6$ RGB color cube
- A 24-level grayscale ramp

We'll also represent this more complicated color space as a variant, but this time, the different tags will have arguments that describe the data available in each case. Note that variants can have multiple arguments, which are separated by *s.

```
type weight = Regular | Bold
type color =
  | Basic of basic_color * weight (* basic colors, regular and bold *)
  | RGB   of int * int * int      (* 6x6x6 color cube *)
  | Gray  of int                  (* 24 grayscale levels *)
```

As before, we can use these introduced tags to construct values of our newly defined type.

```
# [RGB (250,70,70); Basic (Green, Regular)];;
- : color list = [RGB (250, 70, 70); Basic (Green, Regular)]
```

https://doi.org/10.1017/9781009129220.009 Published online by Cambridge University Press

And again, we'll use pattern matching to convert a color to a corresponding integer. In this case, the pattern matching does more than separate out the different cases; it also allows us to extract the data associated with each tag:

```
# let color_to_int = function
    | Basic (basic_color,weight) ->
      let base = match weight with Bold -> 8 | Regular -> 0 in
      base + basic_color_to_int basic_color
    | RGB (r,g,b) -> 16 + b + g * 6 + r * 36
    | Gray i -> 232 + i;;
val color_to_int : color -> int = <fun>
```

Now, we can print text using the full set of available colors:

```
# let color_print color s =
  printf "%s\n" (color_by_number (color_to_int color) s);;
val color_print : color -> string -> unit = <fun>
# color_print (Basic (Red,Bold)) "A bold red!";;
A bold red!
- : unit = ()
# color_print (Gray 4) "A muted gray...";;
A muted gray...
- : unit = ()
```

Variants, Tuples and Parens

Variants with multiple arguments look an awful lot like tuples. Consider the following example of a value of the type `color` we defined earlier.

```
# RGB (200,0,200);;
- : color = RGB (200, 0, 200)
```

It really looks like we've created a 3-tuple and wrapped it with the `RGB` constructor. But that's not what's really going on, as you can see if we create a tuple first and then place it inside the `RGB` constructor.

```
# let purple = (200,0,200);;
val purple : int * int * int = (200, 0, 200)
# RGB purple;;
Line 1, characters 1-11:
Error: The constructor RGB expects 3 argument(s),
       but is applied here to 1 argument(s)
```

We can also create variants that explicitly contain tuples, like this one.

```
# type tupled = Tupled of (int * int);;
type tupled = Tupled of (int * int)
```

The syntactic difference is unfortunately quite subtle, coming down to the extra set of parens around the arguments. But having defined it this way, we can now take the tuple in and out freely.

```
# let of_tuple x = Tupled x;;
val of_tuple : int * int -> tupled = <fun>
# let to_tuple (Tupled x) = x;;
val to_tuple : tupled -> int * int = <fun>
```

https://doi.org/10.1017/9781009129220.009 Published online by Cambridge University Press

If, on the other hand, we define a variant without the parens, then we get the same behavior we got with the `RGB` constructor.

```
# type untupled = Untupled of int * int;;
type untupled = Untupled of int * int
# let of_tuple x = Untupled x;;
Line 1, characters 18-28:
Error: The constructor Untupled expects 2 argument(s),
       but is applied here to 1 argument(s)
# let to_tuple (Untupled x) = x;;
Line 1, characters 14-26:
Error: The constructor Untupled expects 2 argument(s),
       but is applied here to 1 argument(s)
```

Note that, while we can't just grab the tuple as a whole from this type, we can achieve more or less the same ends by explicitly deconstructing and reconstructing the data we need.

```
# let of_tuple (x,y) = Untupled (x,y);;
val of_tuple : int * int -> untupled = <fun>
# let to_tuple (Untupled (x,y)) = (x,y);;
val to_tuple : untupled -> int * int = <fun>
```

The differences between a multi-argument variant and a variant containing a tuple are mostly about performance. A multi-argument variant is a single allocated block in memory, while a variant containing a tuple requires an extra heap-allocated block for the tuple. You can learn more about OCaml's memory representation in Chapter 24 (Memory Representation of Values).

7.1 Catch-All Cases and Refactoring

OCaml's type system can act as a refactoring tool, warning you of places where your code needs to be updated to match an interface change. This is particularly valuable in the context of variants.

Consider what would happen if we were to change the definition of `color` to the following:

```
type color =
  | Basic of basic_color     (* basic colors *)
  | Bold  of basic_color     (* bold basic colors *)
  | RGB   of int * int * int (* 6x6x6 color cube *)
  | Gray  of int             (* 24 grayscale levels *)
```

We've essentially broken out the `Basic` case into two cases, `Basic` and `Bold`, and `Basic` has changed from having two arguments to one. `color_to_int` as we wrote it still expects the old structure of the variant, and if we try to compile that same code again, the compiler will notice the discrepancy:

```
# let color_to_int = function
    | Basic (basic_color,weight) ->
      let base = match weight with Bold -> 8 | Regular -> 0 in
```

https://doi.org/10.1017/9781009129220.009 Published online by Cambridge University Press

```
      base + basic_color_to_int basic_color
    | RGB (r,g,b) -> 16 + b + g * 6 + r * 36
    | Gray i -> 232 + i;;
Line 2, characters 13-33:
Error: This pattern matches values of type 'a * 'b
       but a pattern was expected which matches values of type
    basic_color
```

Here, the compiler is complaining that the `Basic` tag is used with the wrong number of arguments. If we fix that, however, the compiler will flag a second problem, which is that we haven't handled the new `Bold` tag:

```
# let color_to_int = function
    | Basic basic_color -> basic_color_to_int basic_color
    | RGB (r,g,b) -> 16 + b + g * 6 + r * 36
    | Gray i -> 232 + i;;
Lines 1-4, characters 20-24:
Warning 8 [partial-match]: this pattern-matching is not exhaustive.
Here is an example of a case that is not matched:
Bold _
val color_to_int : color -> int = <fun>
```

Fixing this now leads us to the correct implementation:

```
# let color_to_int = function
    | Basic basic_color -> basic_color_to_int basic_color
    | Bold  basic_color -> 8 + basic_color_to_int basic_color
    | RGB (r,g,b) -> 16 + b + g * 6 + r * 36
    | Gray i -> 232 + i;;
val color_to_int : color -> int = <fun>
```

As we've seen, the type errors identified the things that needed to be fixed to complete the refactoring of the code. This is fantastically useful, but for it to work well and reliably, you need to write your code in a way that maximizes the compiler's chances of helping you find the bugs. To this end, a useful rule of thumb is to avoid catch-all cases in pattern matches.

Here's an example that illustrates how catch-all cases interact with exhaustion checks. Imagine we wanted a version of `color_to_int` that works on older terminals by rendering the first 16 colors (the eight `basic_colors` in regular and bold) in the normal way, but renders everything else as white. We might have written the function as follows.

```
# let oldschool_color_to_int = function
    | Basic (basic_color,weight) ->
      let base = match weight with Bold -> 8 | Regular -> 0 in
      base + basic_color_to_int basic_color
    | _ -> basic_color_to_int White;;
val oldschool_color_to_int : color -> int = <fun>
```

If we then applied the same fix we did above, we would have ended up with this.

```
# let oldschool_color_to_int = function
    | Basic basic_color -> basic_color_to_int basic_color
    | _ -> basic_color_to_int White;;
val oldschool_color_to_int : color -> int = <fun>
```

https://doi.org/10.1017/9781009129220.009 Published online by Cambridge University Press

Because of the catch-all case, we'll no longer be warned about missing the `Bold` case. That's why you should beware of catch-all cases: they suppress exhaustiveness checking.

7.2 Combining Records and Variants

The term *algebraic data types* is often used to describe a collection of types that includes variants, records, and tuples. Algebraic data types act as a peculiarly useful and powerful language for describing data. At the heart of their utility is the fact that they combine two different kinds of types: *product types*, like tuples and records, which combine multiple different types together and are mathematically similar to Cartesian products; and *sum types*, like variants, which let you combine multiple different possibilities into one type, and are mathematically similar to disjoint unions.

Algebraic data types gain much of their power from the ability to construct layered combinations of sums and products. Let's see what we can achieve with this by reiterating the `Log_entry` message type that was described in Chapter 6 (Records).

```
module Time_ns = Core.Time_ns
module Log_entry = struct
  type t =
    { session_id: string;
      time: Time_ns.t;
      important: bool;
      message: string;
    }
end
```

This record type combines multiple pieces of data into a single value. In particular, a single `Log_entry.t` has a `session_id` *and* a `time` *and* an `important` flag *and* a `message`. More generally, you can think of record types as conjunctions. Variants, on the other hand, are disjunctions, letting you represent multiple possibilities. To construct an example of where this is useful, we'll first write out the other message types that came along-side `Log_entry`.

```
module Heartbeat = struct
  type t =
    { session_id: string;
      time: Time_ns.t;
      status_message: string;
    }
end
module Logon = struct
  type t =
    { session_id: string;
      time: Time_ns.t;
      user: string;
      credentials: string;
    }
end
```

https://doi.org/10.1017/9781009129220.009 Published online by Cambridge University Press

A variant comes in handy when we want to represent values that could be any of these three types. The `client_message` type below lets you do just that.

```
type client_message = | Logon of Logon.t
                      | Heartbeat of Heartbeat.t
                      | Log_entry of Log_entry.t
```

In particular, a `client_message` is a `Logon` *or* a `Heartbeat` *or* a `Log_entry`. If we want to write code that processes messages generically, rather than code specialized to a fixed message type, we need something like `client_message` to act as one overarching type for the different possible messages. We can then match on the `client_message` to determine the type of the particular message being handled.

You can increase the precision of your types by using variants to represent differences between different cases, and records to represent shared structure. Consider the following function that takes a list of `client_messages` and returns all messages generated by a given user. The code in question is implemented by folding over the list of messages, where the accumulator is a pair of:

- The set of session identifiers for the user that have been seen thus far
- The set of messages so far that are associated with the user

Here's the concrete code:

```
# let messages_for_user user messages =
    let (user_messages,_) =
      List.fold messages ~init:([], Set.empty (module String))
        ~f:(fun ((messages,user_sessions) as acc) message ->
          match message with
          | Logon m ->
            if String.(m.user = user) then
              (message::messages, Set.add user_sessions m.session_id)
            else acc
          | Heartbeat _ | Log_entry _ ->
            let session_id = match message with
              | Logon     m -> m.session_id
              | Heartbeat m -> m.session_id
              | Log_entry m -> m.session_id
            in
            if Set.mem user_sessions session_id then
              (message::messages,user_sessions)
            else acc
        )
    in
    List.rev user_messages;;
val messages_for_user : string -> client_message list ->
    client_message list =
  <fun>
```

We take advantage of the fact that the type of the record `m` is known in the above code, so we don't have to qualify the record fields by the module they come from. *e.g.*, we write `m.user` instead of `m.Logon.user`.

One annoyance of the above code is that the logic for determining the session ID is somewhat repetitive, contemplating each of the possible message types (including the

https://doi.org/10.1017/9781009129220.009 Published online by Cambridge University Press

`Logon` case, which isn't actually possible at that point in the code) and extracting the session ID in each case. This per-message-type handling seems unnecessary, since the session ID works the same way for all message types.

We can improve the code by refactoring our types to explicitly reflect the information that's shared between the different messages. The first step is to cut down the definitions of each per-message record to contain just the information unique to that record:

```
module Log_entry = struct
  type t = { important: bool;
             message: string;
           }
end
module Heartbeat = struct
  type t = { status_message: string; }
end
module Logon = struct
  type t = { user: string;
             credentials: string;
           }
end
```

We can then define a variant type that combines these types:

```
type details =
  | Logon of Logon.t
  | Heartbeat of Heartbeat.t
  | Log_entry of Log_entry.t
```

Separately, we need a record that contains the fields that are common across all messages:

```
module Common = struct
  type t = { session_id: string;
             time: Time_ns.t;
           }
end
```

A full message can then be represented as a pair of a `Common.t` and a `details`. Using this, we can rewrite our preceding example as follows. Note that we add extra type annotations so that OCaml recognizes the record fields correctly. Otherwise, we'd need to qualify them explicitly.

```
# let messages_for_user user (messages : (Common.t * details) list) =
    let (user_messages,_) =
      List.fold messages ~init:([],Set.empty (module String))
        ~f:(fun ((messages,user_sessions) as acc) ((common,details)
    as message) ->
          match details with
          | Logon m ->
            if String.(=) m.user user then
              (message::messages, Set.add user_sessions
    common.session_id)
            else acc
          | Heartbeat _ | Log_entry _ ->
            if Set.mem user_sessions common.session_id then
              (message::messages, user_sessions)
```

https://doi.org/10.1017/9781009129220.009 Published online by Cambridge University Press

```
            else acc
        )
    in
    List.rev user_messages;;
val messages_for_user :
  string -> (Common.t * details) list -> (Common.t * details) list =
    <fun>
```

As you can see, the code for extracting the session ID has been replaced with the simple expression `common.session_id`.

In addition, this design allows us to grab the specific message and dispatch code to handle just that message type. In particular, while we use the type `Common.t * details` to represent an arbitrary message, we can use `Common.t * Logon.t` to represent a logon message. Thus, if we had functions for handling individual message types, we could write a dispatch function as follows:

```
# let handle_message server_state ((common:Common.t), details) =
    match details with
    | Log_entry m -> handle_log_entry server_state (common,m)
    | Logon     m -> handle_logon     server_state (common,m)
    | Heartbeat m -> handle_heartbeat server_state (common,m);;
val handle_message : server_state -> Common.t * details -> unit =
    <fun>
```

And it's explicit at the type level that `handle_log_entry` sees only `Log_entry` messages, `handle_logon` sees only `Logon` messages, etc.

7.2.1 Embedded Records

If we don't need to be able to pass the record types separately from the variant, then OCaml allows us to embed the records directly into the variant.

```
type details =
  | Logon     of { user: string; credentials: string; }
  | Heartbeat of { status_message: string; }
  | Log_entry of { important: bool; message: string; }
```

Even though the type is different, we can write `messages_for_user` in essentially the same way we did before.

```
# let messages_for_user user (messages : (Common.t * details) list) =
    let (user_messages,_) =
      List.fold messages ~init:([],Set.empty (module String))
        ~f:(fun ((messages,user_sessions) as acc) ((common,details)
    as message) ->
          match details with
          | Logon m ->
            if String.(=) m.user user then
              (message::messages, Set.add user_sessions
    common.session_id)
            else acc
          | Heartbeat _ | Log_entry _ ->
            if Set.mem user_sessions common.session_id then
              (message::messages, user_sessions)
```

https://doi.org/10.1017/9781009129220.009 Published online by Cambridge University Press

```
            else acc
        )
    in
    List.rev user_messages;;
val messages_for_user :
  string -> (Common.t * details) list -> (Common.t * details) list =
    <fun>
```

Variants with inline records are both more concise and more efficient than having variants containing references to free-standing record types, because they don't require a separate allocated object for the contents of the variant.

The main downside is the obvious one, which is that an inline record can't be treated as its own free-standing object. And, as you can see below, OCaml will reject code that tries to do so.

```
# let get_logon_contents = function
    | Logon m -> Some m
    | _ -> None;;
Line 2, characters 23-24:
Error: This form is not allowed as the type of the inlined record
    could escape.
```

7.3 Variants and Recursive Data Structures

Another common application of variants is to represent tree-like recursive data structures. We'll show how this can be done by walking through the design of a simple Boolean expression language. Such a language can be useful anywhere you need to specify filters, which are used in everything from packet analyzers to mail clients.

An expression in this language will be defined by the variant `expr`, with one tag for each kind of expression we want to support:

```
type 'a expr =
  | Base  of 'a
  | Const of bool
  | And   of 'a expr list
  | Or    of 'a expr list
  | Not   of 'a expr
```

Note that the definition of the type `expr` is recursive, meaning that an `expr` may contain other `expr`s. Also, `expr` is parameterized by a polymorphic type `'a` which is used for specifying the type of the value that goes under the `Base` tag.

The purpose of each tag is pretty straightforward. `And`, `Or`, and `Not` are the basic operators for building up Boolean expressions, and `Const` lets you enter the constants `true` and `false`.

The `Base` tag is what allows you to tie the `expr` to your application, by letting you specify an element of some base predicate type, whose truth or falsehood is determined by your application. If you were writing a filter language for an email processor, your base predicates might specify the tests you would run against an email, as in the following example:

https://doi.org/10.1017/9781009129220.009 Published online by Cambridge University Press

```
type mail_field = To | From | CC | Date | Subject
type mail_predicate = { field: mail_field;
                        contains: string }
```

Using the preceding code, we can construct a simple expression with `mail_predicate` as its base predicate:

```
# let test field contains = Base { field; contains };;
val test : mail_field -> string -> mail_predicate expr = <fun>
# And [ Or [ test To "doligez"; test CC "doligez" ];
        test Subject "runtime";
      ];;
- : mail_predicate expr =
And
 [Or
   [Base {field = To; contains = "doligez"};
    Base {field = CC; contains = "doligez"}];
  Base {field = Subject; contains = "runtime"}]
```

Being able to construct such expressions isn't enough; we also need to be able to evaluate them. Here's a function for doing just that:

```
# let rec eval expr base_eval =
    (* a shortcut, so we don't need to repeatedly pass [base_eval]
       explicitly to [eval] *)
    let eval' expr = eval expr base_eval in
    match expr with
    | Base  base  -> base_eval base
    | Const bool  -> bool
    | And   exprs -> List.for_all exprs ~f:eval'
    | Or    exprs -> List.exists  exprs ~f:eval'
    | Not   expr  -> not (eval' expr);;
val eval : 'a expr -> ('a -> bool) -> bool = <fun>
```

The structure of the code is pretty straightforward—we're just pattern matching over the structure of the data, doing the appropriate calculation based on which tag we see. To use this evaluator on a concrete example, we just need to write the `base_eval` function, which is capable of evaluating a base predicate.

Another useful operation on expressions is *simplification*, which is the process of taking a boolean expression and reducing it to an equivalent one that is smaller. First, we'll build a few simplifying construction functions that mirror the tags of an `expr`.

The `and_` function below does a few things:

- Reduces the entire expression to the constant `false` if any of the arms of the `And` are themselves `false`.
- Drops any arms of the `And` that there the constant `true`.
- Drops the `And` if it only has one arm.
- If the `And` has no arms, then reduces it to `Const true`.

The code is below.

```
# let and_ l =
    if List.exists l ~f:(function Const false -> true | _ -> false)
    then Const false
```

https://doi.org/10.1017/9781009129220.009 Published online by Cambridge University Press

```
    else
      match List.filter l ~f:(function Const true -> false | _ ->
    true) with
      | [] -> Const true
      | [ x ] -> x
      | l -> And l;;
val and_ : 'a expr list -> 'a expr = <fun>
```

`Or` is the dual of `And`, and as you can see, the code for `or_` follows a similar pattern as that for `and_`, mostly reversing the role of `true` and `false`.

```
# let or_ l =
    if List.exists l ~f:(function Const true -> true | _ -> false)
    then Const true
    else
      match List.filter l ~f:(function Const false -> false | _ ->
    true) with
      | [] -> Const false
      | [x] -> x
      | l -> Or l;;
val or_ : 'a expr list -> 'a expr = <fun>
```

Finally, `not_` just has special handling for constants, applying the ordinary boolean negation function to them.

```
# let not_ = function
    | Const b -> Const (not b)
    | e -> Not e;;
val not_ : 'a expr -> 'a expr = <fun>
```

We can now write a simplification routine that is based on the preceding functions. Note that this function is recursive, in that it applies all of these simplifications in a bottom-up way across the entire expression.

```
# let rec simplify = function
    | Base _ | Const _ as x -> x
    | And l -> and_ (List.map ~f:simplify l)
    | Or l  -> or_  (List.map ~f:simplify l)
    | Not e -> not_ (simplify e);;
val simplify : 'a expr -> 'a expr = <fun>
```

We can now apply this to a Boolean expression and see how good a job it does at simplifying it.

```
# simplify (Not (And [ Or [Base "it's snowing"; Const true];
  Base "it's raining"]));;
- : string expr = Not (Base "it's raining")
```

Here, it correctly converted the `Or` branch to `Const true` and then eliminated the `And` entirely, since the `And` then had only one nontrivial component.

There are some simplifications it misses, however. In particular, see what happens if we add a double negation in.

```
# simplify (Not (And [ Or [Base "it's snowing"; Const true];
  Not (Not (Base "it's raining"))]));;
- : string expr = Not (Not (Not (Base "it's raining")))
```

https://doi.org/10.1017/9781009129220.009 Published online by Cambridge University Press

It fails to remove the double negation, and it's easy to see why. The `not_` function has a catch-all case, so it ignores everything but the one case it explicitly considers, that of the negation of a constant. Catch-all cases are generally a bad idea, and if we make the code more explicit, we see that the missing of the double negation is more obvious:

```
# let not_ = function
    | Const b -> Const (not b)
    | (Base _ | And _ | Or _ | Not _) as e -> Not e;;
val not_ : 'a expr -> 'a expr = <fun>
```

We can of course fix this by simply adding an explicit case for double negation:

```
# let not_ = function
    | Const b -> Const (not b)
    | Not e -> e
    | (Base _ | And _ | Or _ ) as e -> Not e;;
val not_ : 'a expr -> 'a expr = <fun>
```

The example of a Boolean expression language is more than a toy. There's a module very much in this spirit in `Core` called `Blang` (short for "Boolean language"), and it gets a lot of practical use in a variety of applications. The simplification algorithm in particular is useful when you want to use it to specialize the evaluation of expressions for which the evaluation of some of the base predicates is already known.

More generally, using variants to build recursive data structures is a common technique, and shows up everywhere from designing little languages to building complex data structures.

7.4 Polymorphic Variants

In addition to the ordinary variants we've seen so far, OCaml also supports so-called *polymorphic variants*. As we'll see, polymorphic variants are more flexible and syntactically more lightweight than ordinary variants, but that extra power comes at a cost.

Syntactically, polymorphic variants are distinguished from ordinary variants by the leading backtick. And unlike ordinary variants, polymorphic variants can be used without an explicit type declaration:

```
# let three = `Int 3;;
val three : [> `Int of int ] = `Int 3
# let four = `Float 4.;;
val four : [> `Float of float ] = `Float 4.
# let nan = `Not_a_number;;
val nan : [> `Not_a_number ] = `Not_a_number
# [three; four; nan];;
- : [> `Float of float | `Int of int | `Not_a_number ] list =
[`Int 3; `Float 4.; `Not_a_number]
```

As you can see, polymorphic variant types are inferred automatically, and when we combine variants with different tags, the compiler infers a new type that knows about

https://doi.org/10.1017/9781009129220.009 Published online by Cambridge University Press

all of those tags. Note that in the preceding example, the tag name (e.g., `Int) matches the type name (int). This is a common convention in OCaml.

The type system will complain if it sees incompatible uses of the same tag:

```
# let five = `Int "five";;
val five : [> `Int of string ] = `Int "five"
# [three; four; five];;
Line 1, characters 15-19:
Error: This expression has type [> `Int of string ]
       but an expression was expected of type
         [> `Float of float | `Int of int ]
       Types for tag `Int are incompatible
```

The > at the beginning of the variant types above is critical because it marks the types as being open to combination with other variant types. We can read the type [> `Float of float | `Int of int] as describing a variant whose tags include `Float of float and `Int of int, but may include more tags as well. In other words, you can roughly translate > to mean: "these tags or more."

OCaml will in some cases infer a variant type with <, to indicate "these tags or less," as in the following example:

```
# let is_positive = function
    | `Int   x -> x > 0
    | `Float x -> Float.(x > 0.);;
val is_positive : [< `Float of float | `Int of int ] -> bool = <fun>
```

The < is there because is_positive has no way of dealing with values that have tags other than `Float of float or `Int of int, but can handle types that have either or both of those two tags.

We can think of these < and > markers as indications of upper and lower bounds on the tags involved. If the same set of tags are both an upper and a lower bound, we end up with an *exact* polymorphic variant type, which has neither marker. For example:

```
# let exact = List.filter ~f:is_positive [three;four];;
val exact : [ `Float of float | `Int of int ] list = [`Int 3; `Float
    4.]
```

Perhaps surprisingly, we can also create polymorphic variant types that have distinct upper and lower bounds. Note that Ok and Error in the following example come from the Result.t type from Base.

```
# let is_positive = function
    | `Int   x -> Ok (x > 0)
    | `Float x -> Ok Float.O.(x > 0.)
    | `Not_a_number -> Error "not a number";;
val is_positive :
  [< `Float of float | `Int of int | `Not_a_number ] -> (bool, string)
    result =
  <fun>
# List.filter [three; four] ~f:(fun x ->
  match is_positive x with Error _ -> false | Ok b -> b);;
- : [< `Float of float | `Int of int | `Not_a_number > `Float `Int ]
    list =
[`Int 3; `Float 4.]
```

https://doi.org/10.1017/9781009129220.009 Published online by Cambridge University Press

Here, the inferred type states that the tags can be no more than `Float, `Int, and `Not_a_number, and must contain at least `Float and `Int. As you can already start to see, polymorphic variants can lead to fairly complex inferred types.

Polymorphic Variants and Catch-All Cases

As we saw with the definition of is_positive, a match expression can lead to the inference of an upper bound on a variant type, limiting the possible tags to those that can be handled by the match. If we add a catch-all case to our match expression, we end up with a type with a lower bound.

```
# let is_positive_permissive = function
    | `Int   x -> Ok Int.(x > 0)
    | `Float x -> Ok Float.(x > 0.)
    | _ -> Error "Unknown number type";;
val is_positive_permissive :
  [> `Float of float | `Int of int ] -> (bool, string) result = <fun>
# is_positive_permissive (`Int 0);;
- : (bool, string) result = Ok false
# is_positive_permissive (`Ratio (3,4));;
- : (bool, string) result = Error "Unknown number type"
```

Catch-all cases are error-prone even with ordinary variants, but they are especially so with polymorphic variants. That's because you have no way of bounding what tags your function might have to deal with. Such code is particularly vulnerable to typos. For instance, if code that uses is_positive_permissive passes in Float misspelled as Floot, the erroneous code will compile without complaint.

```
# is_positive_permissive (`Floot 3.5);;
- : (bool, string) result = Error "Unknown number type"
```

With ordinary variants, such a typo would have been caught as an unknown tag. As a general matter, one should be wary about mixing catch-all cases and polymorphic variants.

7.4.1 Example: Terminal Colors Redux

To see how to use polymorphic variants in practice, we'll return to terminal colors. Imagine that we have a new terminal type that adds yet more colors, say, by adding an alpha channel so you can specify translucent colors. We could model this extended set of colors as follows, using an ordinary variant:

```
type extended_color =
  | Basic of basic_color * weight  (* basic colors, regular and bold
    *)
  | RGB   of int * int * int       (* 6x6x6 color space *)
  | Gray  of int                   (* 24 grayscale levels *)
  | RGBA  of int * int * int * int (* 6x6x6x6 color space *)
```

We want to write a function extended_color_to_int that works like color_to_int for all of the old kinds of colors, with new logic only for handling colors that include an alpha channel. One might try to write such a function as follows.

https://doi.org/10.1017/9781009129220.009 Published online by Cambridge University Press

```
# let extended_color_to_int = function
    | RGBA (r,g,b,a) -> 256 + a + b * 6 + g * 36 + r * 216
    | (Basic _ | RGB _ | Gray _) as color -> color_to_int color;;
Line 3, characters 59-64:
Error: This expression has type extended_color
       but an expression was expected of type color
```

The code looks reasonable enough, but it leads to a type error because `extended_color` and `color` are in the compiler's view distinct and unrelated types. The compiler doesn't, for example, recognize any equality between the `Basic` tag in the two types.

What we want to do is to share tags between two different variant types, and polymorphic variants let us do this in a natural way. First, let's rewrite `basic_color_to_int` and `color_to_int` using polymorphic variants. The translation here is pretty straightforward:

```
# let basic_color_to_int = function
    | `Black -> 0 | `Red     -> 1 | `Green -> 2 | `Yellow -> 3
    | `Blue  -> 4 | `Magenta -> 5 | `Cyan  -> 6 | `White  -> 7;;
val basic_color_to_int :
  [< `Black | `Blue | `Cyan | `Green | `Magenta | `Red | `White |
    `Yellow ] ->
  int = <fun>
# let color_to_int = function
    | `Basic (basic_color,weight) ->
      let base = match weight with `Bold -> 8 | `Regular -> 0 in
      base + basic_color_to_int basic_color
    | `RGB (r,g,b) -> 16 + b + g * 6 + r * 36
    | `Gray i -> 232 + i;;
val color_to_int :
  [< `Basic of
       [< `Black
        | `Blue
        | `Cyan
        | `Green
        | `Magenta
        | `Red
        | `White
        | `Yellow ] *
       [< `Bold | `Regular ]
   | `Gray of int
   | `RGB of int * int * int ] ->
  int = <fun>
```

Now we can try writing `extended_color_to_int`. The key issue with this code is that `extended_color_to_int` needs to invoke `color_to_int` with a narrower type, i.e., one that includes fewer tags. Written properly, this narrowing can be done via a pattern match. In particular, in the following code, the type of the variable `color` includes only the tags `` `Basic ``, `` `RGB ``, and `` `Gray ``, and not `` `RGBA ``:

```
# let extended_color_to_int = function
    | `RGBA (r,g,b,a) -> 256 + a + b * 6 + g * 36 + r * 216
    | (`Basic _ | `RGB _ | `Gray _) as color -> color_to_int color;;
val extended_color_to_int :
  [< `Basic of
```

https://doi.org/10.1017/9781009129220.009 Published online by Cambridge University Press

```
        [< `Black
         | `Blue
         | `Cyan
         | `Green
         | `Magenta
         | `Red
         | `White
         | `Yellow ] *
        [< `Bold | `Regular ]
    | `Gray of int
    | `RGB of int * int * int
    | `RGBA of int * int * int * int ] ->
  int = <fun>
```

The preceding code is more delicately balanced than one might imagine. In particular, if we use a catch-all case instead of an explicit enumeration of the cases, the type is no longer narrowed, and so compilation fails:

```
# let extended_color_to_int = function
    | `RGBA (r,g,b,a) -> 256 + a + b * 6 + g * 36 + r * 216
    | color -> color_to_int color;;
Line 3, characters 29-34:
Error: This expression has type [> `RGBA of int * int * int * int ]
       but an expression was expected of type
         [< `Basic of
              [< `Black
               | `Blue
               | `Cyan
               | `Green
               | `Magenta
               | `Red
               | `White
               | `Yellow ] *
              [< `Bold | `Regular ]
         | `Gray of int
         | `RGB of int * int * int ]
       The second variant type does not allow tag(s) `RGBA
```

Let's consider how we might turn our code into a proper library with an implementation in an `ml` file and an interface in a separate `mli`, as we saw in Chapter 5 (Files, Modules, and Programs). Let's start with the `mli`.

```
open Base

type basic_color =
  [ `Black   | `Blue | `Cyan  | `Green
  | `Magenta | `Red  | `White | `Yellow ]

type color =
  [ `Basic of basic_color * [ `Bold | `Regular ]
  | `Gray of int
  | `RGB  of int * int * int ]

type extended_color =
  [ color
  | `RGBA of int * int * int * int ]
```

https://doi.org/10.1017/9781009129220.009 Published online by Cambridge University Press

```
val color_to_int          : color -> int
val extended_color_to_int : extended_color -> int
```

Here, `extended_color` is defined as an explicit extension of `color`. Also, notice that we defined all of these types as exact variants. We can implement this library as follows.

```
open Base

type basic_color =
  [ `Black   | `Blue | `Cyan  | `Green
  | `Magenta | `Red  | `White | `Yellow ]

type color =
  [ `Basic of basic_color * [ `Bold | `Regular ]
  | `Gray of int
  | `RGB  of int * int * int ]

type extended_color =
  [ color
  | `RGBA of int * int * int * int ]

let basic_color_to_int = function
  | `Black -> 0 | `Red     -> 1 | `Green -> 2 | `Yellow -> 3
  | `Blue  -> 4 | `Magenta -> 5 | `Cyan  -> 6 | `White  -> 7

let color_to_int = function
  | `Basic (basic_color,weight) ->
    let base = match weight with `Bold -> 8 | `Regular -> 0 in
    base + basic_color_to_int basic_color
  | `RGB (r,g,b) -> 16 + b + g * 6 + r * 36
  | `Gray i -> 232 + i

let extended_color_to_int = function
  | `RGBA (r,g,b,a) -> 256 + a + b * 6 + g * 36 + r * 216
  | `Grey x -> 2000 + x
  | (`Basic _ | `RGB _ | `Gray _) as color -> color_to_int color
```

In the preceding code, we did something funny to the definition of `extended_color_to_int` that highlights some of the downsides of polymorphic variants. In particular, we added some special-case handling for the color gray, rather than using `color_to_int`. Unfortunately, we misspelled `Gray` as `Grey`. This is exactly the kind of error that the compiler would catch with ordinary variants, but with polymorphic variants, this compiles without issue. All that happened was that the compiler inferred a wider type for `extended_color_to_int`, which happens to be compatible with the narrower type that was listed in the `mli`. As a result, this library builds without error.

```
$ dune build @all
```

If we add an explicit type annotation to the code itself (rather than just in the `mli`), then the compiler has enough information to warn us:

```
let extended_color_to_int : extended_color -> int = function
  | `RGBA (r,g,b,a) -> 256 + a + b * 6 + g * 36 + r * 216
```

https://doi.org/10.1017/9781009129220.009 Published online by Cambridge University Press

```
  | `Grey x -> 2000 + x
  | (`Basic _ | `RGB _ | `Gray _) as color -> color_to_int color
```

In particular, the compiler will complain that the `Grey case is unused:

```
$ dune build @all
File "terminal_color.ml", line 30, characters 4-11:
30 |   | `Grey x -> 2000 + x
         ^^^^^^^
Error: This pattern matches values of type [? `Grey of 'a ]
       but a pattern was expected which matches values of type
    extended_color
      The second variant type does not allow tag(s) `Grey
[1]
```

Once we have type definitions at our disposal, we can revisit the question of how we write the pattern match that narrows the type. In particular, we can explicitly use the type name as part of the pattern match, by prefixing it with a #:

```
let extended_color_to_int : extended_color -> int = function
  | `RGBA (r,g,b,a) -> 256 + a + b * 6 + g * 36 + r * 216
  | #color as color -> color_to_int color
```

This is useful when you want to narrow down to a type whose definition is long, and you don't want the verbosity of writing the tags down explicitly in the match.

7.4.2 When to Use Polymorphic Variants

At first glance, polymorphic variants look like a strict improvement over ordinary variants. You can do everything that ordinary variants can do, plus it's more flexible and more concise. What's not to like?

In reality, regular variants are the more pragmatic choice most of the time. That's because the flexibility of polymorphic variants comes at a price. Here are some of the downsides:

Complexity The typing rules for polymorphic variants are a lot more complicated than they are for regular variants. This means that heavy use of polymorphic variants can leave you scratching your head trying to figure out why a given piece of code did or didn't compile. It can also lead to absurdly long and hard to decode error messages. Indeed, concision at the value level is often balanced out by more verbosity at the type level.

Error-finding Polymorphic variants are type-safe, but the typing discipline that they impose is, by dint of its flexibility, less likely to catch bugs in your program.

Efficiency This isn't a huge effect, but polymorphic variants are somewhat heavier than regular variants, and OCaml can't generate code for matching on polymorphic variants that is quite as efficient as what it generated for regular variants.

All that said, polymorphic variants are still a useful and powerful feature, but it's worth understanding their limitations and how to use them sensibly and modestly.

Probably the safest and most common use case for polymorphic variants is where

https://doi.org/10.1017/9781009129220.009 Published online by Cambridge University Press

ordinary variants would be sufficient but are syntactically too heavyweight. For example, you often want to create a variant type for encoding the inputs or outputs to a function, where it's not worth declaring a separate type for it. Polymorphic variants are very useful here, and as long as there are type annotations that constrain these to have explicit, exact types, this tends to work well.

Variants are most problematic exactly where you take full advantage of their power; in particular, when you take advantage of the ability of polymorphic variant types to overlap in the tags they support. This ties into OCaml's support for subtyping. As we'll discuss further when we cover objects in Chapter 13 (Objects), subtyping brings in a lot of complexity, and most of the time, that's complexity you want to avoid.

https://doi.org/10.1017/9781009129220.009 Published online by Cambridge University Press