



Buy in [print](#) and [eBook](#).

Table of Contents

Prologue

I. Language Concepts

1. A Guided Tour
2. Variables and Functions
3. Lists and Patterns
4. Files, Modules, and Programs
5. Records
6. **Variants**
7. Error Handling
8. Imperative Programming
9. Functors
10. First-Class Modules
11. Objects
12. Classes

II. Tools and Techniques

III. The Runtime System

Index

Chapter 6. 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.[0 comments](#)

The basic syntax of a variant type declaration is as follows:[0 comments](#)

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

Syntax * variants/variant.syntax * all code

Each row essentially represents a case of the variant. Each case has an associated tag and may optionally have a sequence of fields, where each field has a specified type.[0 comments](#)

Let's consider a concrete example of how variants can be useful. Almost all terminals support a set of eight basic colors, and we can represent those colors using 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.**[0 comments](#)

```
# type basic_color =  
  | Black | Red | Green | Yellow | Blue | Magenta | Cyan | White ;;  
type basic_color =  
  Black  
  | Red  
  | Green  
  | Yellow  
  | Blue  
  | Magenta  
  | Cyan  
  | White  
# Cyan ;;  
- : basic_color = Cyan  
# [Blue; Magenta; Red] ;;  
- : basic_color list = [Blue; Magenta; Red]
```

OCaml Utop * variants/main.topscript * all code

The following function uses pattern matching to convert a `basic_color` to a corresponding integer. The exhaustiveness checking on pattern matches means that the compiler will warn us if we miss a color:[0 comments](#)

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

OCaml Utop * variants/main.topscript , continued (part 1) * all code

Using the preceding function, we can generate escape codes to change the color of a given string displayed in a terminal:[0 comments](#)

```
# let color_by_number number text =  
  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!
```

OCaml Utop * variants/main-2.rawscript * all code

On most terminals, that word "Blue" will be rendered in blue.[0 comments](#)

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 a simple enumeration. 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:[0 comments](#)

- The eight basic colors, in regular and bold versions[0 comments](#)
- A $6 \times 6 \times 6$ RGB color cube[0 comments](#)
- A 24-level grayscale ramp[0 comments](#)

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:[0 comments](#)

```
# 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 *)
;;
type weight = Regular | Bold
type color =
  Basic of basic_color * weight
  | RGB of int * int * int
  | Gray of int
# [RGB (250,70,70); Basic (Green, Regular)];;
- : color list = [RGB (250, 70, 70); Basic (Green, Regular)]
```

OCaml Utop * variants/main.topscript , continued (part 3) * all code

Once again, we'll use pattern matching to convert a color to a corresponding integer. But 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:[0 comments](#)

```
# 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>
```

OCaml Utop * variants/main.topscript , continued (part 4) * all code

Now, we can print text using the full set of available colors:[0 comments](#)

```
# 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!
# color_print (Gray 4) "A muted gray...";;
A muted gray...
```

OCaml Utop * variants/main-5.rawscript * all code

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.[0 comments](#)

Consider what would happen if we were to change the definition of `color` to the following:[0 comments](#)

```
# 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 *)
;;
type color =
  Basic of basic_color
```

```

| Bold of basic_color
| RGB of int * int * int
| Gray of int

```

OCaml Utop * variants/catch_all.topscript , continued (part 1) * all code

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:[0 comments](#)

```

# 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 ;;

Characters 34-60:
Error: This pattern matches values of type 'a * 'b
      but a pattern was expected which matches values of type basic_color

```

OCaml Utop * variants/catch_all.topscript , continued (part 2) * all code

Here, the compiler is complaining that the `Basic` tag is used with the wrong number of arguments. If we fix that, however, the compiler flag will flag a second problem, which is that we haven't handled the new `Bold` tag:[0 comments](#)

```

# 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 ;;

Characters 19-154:
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
Bold _val color_to_int : color -> int = <fun>

```

OCaml Utop * variants/catch_all.topscript , continued (part 3) * all code

Fixing this now leads us to the correct implementation:[0 comments](#)

```

# 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>

```

OCaml Utop * variants/catch_all.topscript , continued (part 4) * all code

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**.[0 comments](#)

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:[0 comments](#)

```

# 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;;

Characters 44-70:
Error: This pattern matches values of type 'a * 'b
      but a pattern was expected which matches values of type basic_color

```

OCaml Utop * variants/catch_all.topscript , continued (part 5) * all code

But because the catch-all case encompasses all possibilities, the type system will no longer warn us that we have missed the new `Bold` case when we change the type to include it. We can get this check back by avoiding the catch-all case, and instead being explicit about the tags that are ignored.[0 comments](#)

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. [0 comments](#)

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 revisiting the logging server types that were described in [Chapter 5, Records](#). We'll start by reminding ourselves of the definition of `Log_entry.t`. [0 comments](#)

```
# module Log_entry = struct
  type t =
    { session_id: string;
      time: Time.t;
      important: bool;
      message: string;
    }
end
;;
module Log_entry :
sig
  type t = {
    session_id : string;
    time : Time.t;
    important : bool;
    message : string;
  }
end
```

OCaml Utop * variants/logger.topscript , continued (part 1) * all code

This record type combines multiple pieces of data into one 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, as in the following example: [0 comments](#)

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

OCaml Utop * variants/logger.topscript , continued (part 2) * all code

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 dealt with. [0 comments](#)

You can increase the precision of your types by using variants to represent differences between types, 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: [0 comments](#)

- The set of session identifiers for the user that have been seen thus far [0 comments](#)
- The set of messages so far that are associated with the user [0 comments](#)

Here's the concrete code: [0 comments](#)

```
# let messages_for_user user messages =
  let (user_messages, _) =
    List.fold messages ~init:([],String.Set.empty)
    ~f:(fun ((messages,user_sessions) as acc) message ->
      match message with
      | Logon m ->
        if m.Logon.user = user then
```

```

        (message::messages, Set.add user_sessions m.Logon.session_id)
    else acc
  | Heartbeat _ | Log_entry _ ->
    let session_id = match message with
    | Logon m -> m.Logon.session_id
    | Heartbeat m -> m.Heartbeat.session_id
    | Log_entry m -> m.Log_entry.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>

```

OCaml Utop * variants/logger.topscript , continued (part 3) * all code

There's one awkward part of the preceding code, which is the logic that determines the session ID. The code is somewhat repetitive, contemplating each of the possible message types (including the `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 of the message types.[0 comments](#)

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:[0 comments](#)

```

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

```

OCaml Utop * variants/logger.topscript , continued (part 4) * all code

We can then define a variant type that combines these types:[0 comments](#)

```

# type details =
| Logon of Logon.t
| Heartbeat of Heartbeat.t
| Log_entry of Log_entry.t
;;
type details =
  Logon of Logon.t
| Heartbeat of Heartbeat.t
| Log_entry of Log_entry.t

```

OCaml Utop * variants/logger.topscript , continued (part 5) * all code

Separately, we need a record that contains the fields that are common across all messages:[0 comments](#)

```

# module Common = struct
  type t = { session_id: string;
             time: Time.t;
           }
end ;;
module Common : sig type t = { session_id : string; time : Time.t; } end

```

OCaml Utop * variants/logger.topscript , continued (part 6) * all code

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:[0 comments](#)

```

# let messages_for_user user messages =

```

```

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

```

OCaml Utop * variants/logger.topscript , continued (part 7) * all code

As you can see, the code for extracting the session ID has been replaced with the simple expression `common.Common.session_id`. [0 comments](#)

In addition, this design allows us to essentially downcast to the specific message type once we know what it is and then 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: [0 comments](#)

```

# let handle_message server_state (common,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)
;;

```

Characters 95-111:
 Error: Unbound value handle_log_entry

OCaml Utop * variants/logger.topscript , continued (part 8) * all code

And it's explicit at the type level that `handle_log_entry` sees only `Log_entry` messages, `handle_logon` sees only `Logon` messages, etc. [0 comments](#)

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. [0 comments](#)

An expression in this language will be defined by the variant `expr`, with one tag for each kind of expression we want to support: [0 comments](#)

```

# type 'a expr =
  | Base of 'a
  | Const of bool
  | And of 'a expr list
  | Or of 'a expr list
  | Not of 'a expr
;;
type 'a expr =
  Base of 'a
  | Const of bool
  | And of 'a expr List
  | Or of 'a expr List
  | Not of 'a expr

```

OCaml Utop * variants/clang.topscript * all code

Note that the definition of the type `expr` is recursive, meaning that a `expr` may contain other `exprs`. 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. [0 comments](#)

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`. [0 comments](#)

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:[0 comments](#)

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

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

OCaml Utop * variants/clang.topscript , continued (part 1) * all code

Using the preceding code, we can construct a simple expression with `mail_predicate` as its base predicate:[0 comments](#)

```
# 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"}]
```

OCaml Utop * variants/clang.topscript , continued (part 2) * all code

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:[0 comments](#)

```
# 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>
```

OCaml Utop * variants/clang.topscript , continued (part 3) * all code

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.[0 comments](#)

Another useful operation on expressions is simplification. The following is a set of simplifying construction functions that mirror the tags of an `expr`:[0 comments](#)

```
# let and_1 =
  if List.mem 1 (Const false) then Const false
  else
    match List.filter 1 ~f:(<>) (Const true) with
    | [] -> Const true
    | [ x ] -> x
    | 1 -> And 1

let or_1 =
  if List.mem 1 (Const true) then Const true
  else
    match List.filter 1 ~f:(<>) (Const false) with
    | [] -> Const false
    | [ x ] -> x
    | 1 -> Or 1

let not_ = function
  | Const b -> Const (not b)
  | e -> Not e
;;
```

```
val and_ : 'a expr list -> 'a expr = <fun>
val or_ : 'a expr list -> 'a expr = <fun>
val not_ : 'a expr -> 'a expr = <fun>
```

OCaml Utop * variants/blank.topscript , continued (part 4) * all code

We can now write a simplification routine that is based on the preceding functions.[0 comments](#)

```
# 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>
```

OCaml Utop * variants/blank.topscript , continued (part 5) * all code

We can apply this to a Boolean expression and see how good a job it does at simplifying it:[0 comments](#)

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

OCaml Utop * variants/blank.topscript , continued (part 6) * all code

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.[0 comments](#)

There are some simplifications it misses, however. In particular, see what happens if we add a double negation in:[0 comments](#)

```
# 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")))
```

OCaml Utop * variants/blank.topscript , continued (part 7) * all code

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:[0 comments](#)

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

OCaml Utop * variants/blank.topscript , continued (part 8) * all code

We can of course fix this by simply adding an explicit case for double negation:[0 comments](#)

```
# 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>
```

OCaml Utop * variants/blank.topscript , continued (part 9) * all code

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.[0 comments](#)

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.[0 comments](#)

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.[0 comments](#)

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:[0 comments](#)

```
# 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 ]
```

OCaml Utop * variants/main.topscript , continued (part 6) * all code

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 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.[0 comments](#)

The type system will complain if it sees incompatible uses of the same tag:[0 comments](#)

```
# let five = `Int "five";;
val five : [> `Int of string ] = `Int "five"
# [three; four; five];;
Characters 14-18:
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
```

OCaml Utop * variants/main.topscript , continued (part 7) * all code

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 `[> `Int of string | `Float of float]` as describing a variant whose tags include ``Int` of `string` and ``Float` of `float`, but may include more tags as well. In other words, you can roughly translate `>` to mean: "these tags or more."[0 comments](#)

OCaml will in some cases infer a variant type with `<`, to indicate "these tags or less," as in the following example:[0 comments](#)

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

OCaml Utop * variants/main.topscript , continued (part 8) * all code

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`.[0 comments](#)

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:[0 comments](#)

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

OCaml Utop * variants/main.topscript , continued (part 9) * all code

Perhaps surprisingly, we can also create polymorphic variant types that have different upper and lower bounds. Note that `Ok` and `Error` in the following example come from the `Result.t` type from `Core`:[0 comments](#)

```
# let is_positive = function
| `Int x -> Ok (x > 0)
| `Float x -> Ok (x > 0.)
| `Not_a_number -> Error "not a number";;
val is_positive :
```

```
[< `Float of float | `Int of int | `Not_a_number ] ->
(bool, string) Result.t = <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.]
```

OCaml Utop * variants/main.topscript , continued (part 10) * all code

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.[0 comments](#)

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:[0 comments](#)

```
# 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 *)
;;
type extended_color =
  Basic of basic_color * weight
| RGB of int * int * int
| Gray of int
| RGBA of int * int * int * int
```

OCaml Utop * variants/main.topscript , continued (part 11) * all code

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.[0 comments](#)

```
# 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
;;
Characters 154-159:
Error: This expression has type extended_color
      but an expression was expected of type color
```

OCaml Utop * variants/main.topscript , continued (part 12) * all code

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.[0 comments](#)

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:[0 comments](#)

```
# 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
;;
val basic_color_to_int :
[< `Black | `Blue | `Cyan | `Green | `Magenta | `Red | `White | `Yellow ] ->
int = <fun>
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>

```

OCaml Utop * variants/main.topscript , continued (part 13) * all code

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`:[0 comments](#)

```

# 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
    [< `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>

```

OCaml Utop * variants/main.topscript , continued (part 14) * all code

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:[0 comments](#)

```

# let extended_color_to_int = function
  | `RGBA (r,g,b,a) -> 256 + a + b * 6 + g * 36 + r * 216
  | color -> color_to_int color
;;
Characters 125-130:
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

```

OCaml Utop * variants/main.topscript , continued (part 15) * all code

Polymorphic Variants and Catch-all Cases

As we saw with the definition of `is_positive`, a `match` statement 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` statement, we end up with a type with a lower bound:[0 comments](#)

```

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

```

```
# is_positive_permissive (`Int 0);;
- : (bool, string) Result.t = Ok false
# is_positive_permissive (`Ratio (3,4));;
- : (bool, string) Result.t = Error "Unknown number type"
```

OCaml Utop * variants/main.topscript , continued (part 16) * all code

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:[0 comments](#)

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

OCaml Utop * variants/main.topscript , continued (part 17) * all code

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.[0 comments](#)

Let's consider how we might turn our code into a proper library with an implementation in an `m1` file and an interface in a separate `mli`, as we saw in [Chapter 4, Files, Modules, and Programs](#). Let's start with the `mli`:[0 comments](#)

```
open Core.Std

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 ]

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

OCaml * variants-termcol/terminal_color.mli * all code

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:[0 comments](#)

```
open Core.Std

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
```

OCaml * variants-termcol/terminal_color.ml * all code

In the preceding code, we did something funny to the definition of `extended_color_to_int` that underlines 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.0 comments](#)

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: [0 comments](#)

```
open Core.Std

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
```

OCaml * variants-termcol-annotated/terminal_color.ml * all code

In particular, the compiler will complain that the ``Grey` case is unused: [0 comments](#)

```
$ corebuild terminal_color.native
File "terminal_color.ml", line 30, characters 4-11:
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
Command exited with code 2.
```

Terminal * variants-termcol-annotated/build.out * all code

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 `#`: [0 comments](#)

```
open Core.Std

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
```

OCaml * variants-termcol-fixed/terminal_color.ml * all code

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.[0 comments](#)

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?[0 comments](#)

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:[0 comments](#)

Complexity

As we've seen, 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.[0 comments](#)

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.[0 comments](#)

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.[0 comments](#)

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.[0 comments](#)

Probably the safest and most common use case for polymorphic variants is where 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.[0 comments](#)

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 11, *Objects*](#), subtyping brings in a lot of complexity, and most of the time, that's complexity you want to avoid.[0 comments](#)

[< Previous](#)

[Next >](#)