Extracted from:

Web Development with ReasonML

Type-Safe, Functional Programming for JavaScript Developers

The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

Web Development with ReasonML

Type-Safe, Functional Programming for JavaScript Developers

J. David Eisenberg

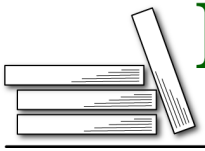
edited by Andrea Stewart

Web Development with ReasonML

Type-Safe, Functional Programming for JavaScript Developers

J. David Eisenberg

The Pragmatic Bookshelf
Raleigh, North Carolina



Our Pragmatic books, screencasts, and audio books can help you and your team create better software and have more fun. Visit us at *https://pragprog.com*.

The team that produced this book includes:

Publisher: Andy Hunt
VP of Operations: Janet Furlow
Managing Editor: Susan Conant
Development Editor: Andrea Stewart
Copy Editor: Sean Dennis
Indexing: Potomac Indexing, LLC
Layout: Gilson Graphics

For sales, volume licensing, and support, please contact *support@pragprog.com*.

For international rights, please contact *rights@pragprog.com*.

ISBN-13: 978-1-68050-633-4
Book version: P1.0—April 2019

Creating Variant Data Types

Here's where ReasonML's type system begins to show some of its power. Let's say we want a data type to represent shirt sizes: Small, Medium, Large, and XLarge (extra-large). We could use an alias for the string type, but it wouldn't keep us from doing things like this:

datatypes/src/StringSizes.re

```
type shirtSize = string;

let mySize = "Medium";
let otherSize = "Large";
let wrongSize = "M";
```

ReasonML lets us create a data type that allows only valid values with a *data type constructor*, which, as its name implies, tells us how to construct a value of that particular data type. This is called a *variant data type*, as we're specifying the various values the data type can have:

datatypes/src/ShirtSizes.re

```
type shirtSize =
  | Small
  | Medium
  | Large
  | XLarge;
```

Constructor names *must* begin with a capital letter. We can bind shirtSize values to variables. The first example has its type annotated:

datatypes/src/ShirtSizes.re

```
let mySize: shirtSize = Medium;
let otherSize = Large;
```

The constructors for a variant data type give you all the possible values. But they're *not* strings! Doing the following:

```
let badSize: shirtSize = "Medium";
```

Gives us this error:

```
We've found a bug for you!
/path/to/code/datatypes/src/ShirtSizes.re 16:26-33

15 │
16 │ let badSize: shirtSize = "Medium";

This has type:
  string
But somewhere wanted:
  shirtSize
```

If we try to create a shirtSize binding with an illegal value:

```
let badSize: shirtSize = M;
```

The compiler tells us that we've used a value that isn't in our data type:

```
We've found a bug for you!
/path/to/code/datatypes/src/ShirtSizes.re 16:26

15 │
16 │ let badSize: shirtSize = M;

This variant expression is expected to have type shirtSize
The constructor M doesn't belong to type shirtSize
```

Using Variant Data Types

Let's say that a small shirt costs $11.00, a medium costs $12.50, a large costs $14.00, and an extra-large costs $16.00. We can write a function to return the price of a shirt given its size:

datatypes/src/ShirtSizes.re

```
let priceIf = (size: shirtSize) : float => {
  if (size === Small) {
    11.00
  } else if (size === Medium) {
    12.50
  } else if (size === Large) {
    14.00
  } else {
    16.00
  }
};

Js.log(priceIf(mySize)); /* output: 12.5 */
Js.log(priceIf(otherSize)); /* output: 14 */
```

But it's much more common in ReasonML to use a switch expression to *pattern match* the size:

datatypes/src/ShirtSizes.re

```
let price = (size: shirtSize) : float => {
  switch (size) {
    | Small => 11.00
    | Medium => 12.50
    | Large => 14.00
    | XLarge => 16.00
  }
};

Js.log(price(mySize)); /* output: 12.5 */
Js.log(price(otherSize)); /* output: 14 */
```

Each of the variants (patterns) is preceded by a vertical bar | and followed by a thick arrow =>, which is followed by the expression to yield for that variant. You can think of the vertical bar as introducing an alternative to match to. ReasonML attempts to match the value size with each of the patterns in the order given. When we do a pattern match on a variant data type, we *must* account for all the variants. If we were to leave off the pattern match for XLarge, we would get this error:

```
  Warning number 8
  28 |
  29 | let price = (size: shirtSize) : float => {
  30 |   switch (size) {
   . | ...
  34 |   }
  35 | };
  36 |
  You forgot to handle a possible value here, for example:
XLarge
```

Let's use switch to write a function that converts a shirtSize value to a string giving the abbreviation for the sizes:

datatypes/src/ShirtSizes.re

```
let stringOfShirtSize = (size: shirtSize) : string => {
  switch (size) {
    | Small => "S"
    | Medium => "M"
    | Large => "L"
    | XLarge => "XL"
  };
};

Js.log(stringOfShirtSize(mySize)); /* output: M */
```

We absolutely need the stringOfShirtSize() function. Consider this code:

datatypes/src/PrintType.re

```
type shirtSize =
  | Small
  | Medium
  | Large
  | XLarge;

let mySize = Medium;
Js.log2("Size is", mySize);
```

Here's what you get when you run it:

```
you@computer:~/book_projects/datatypes> node src/PrintType.bs.js
Size is 1
```

What's going on here?! Why do we get a number? The answer is that all of ReasonML's type checking and manipulation is done entirely at compile time. Once the types are checked, ReasonML is free to use any internal form it likes

to represent the types. In this case, it is optimized into numeric form at runtime, as we can see in the JavaScript code that was generated:

datatypes/src/PrintType.bs.js

```
// Generated by BUCKLESCRIPT VERSION 5.0.0-dev.4, PLEASE EDIT WITH CARE
'use strict';

console.log("Size is", /* Medium */1);

var mySize = /* Medium */1;

exports.mySize = mySize;
/*  Not a pure module */
```

The moral of the story: ReasonML's data types exist only at compile time. If you want to display the value in a readable form, you must provide a function to convert the type to a string.

We'll also want a function that converts a `string` parameter, an abbreviation for the size, to a `shirtSize` value. But we have a problem: what happens if someone gives us a bad string, such as `"N"` or `"Medium"`? If `switch` requires us to write out all possible values, how do we handle all possible strings? Luckily, `switch` is provided with a catch-all pattern, `_` (underscore), which stands for "any case that hasn't been matched yet."

datatypes/src/ShirtSizes.re

```
let shirtSizeOfString = (str: string) : shirtSize => {
  switch (str) {
    | "S" => Small
    | "M" => Medium
    | "L" => Large
    | "XL" => XLarge
    | _ => Medium
  }
};
```

Our approach in this code is to throw our hands up in the air and say, "If we can't figure out what you want, we'll give you `Medium`." If you aren't thrilled with this, don't worry—we'll find a better way to handle this later in the chapter.

Creating Variant Data Types with Parameters

Shirt sizes don't end with extra-large. There are double, triple, and even quadruple extra-large, usually abbreviated as XXL, XXXL, and XXXXL. Parameterized types let us specify a parameter for the constructor. In our case, we want the parameter to tell us how many `X`s are on the shirt size.

Here's a parameterized version of the shirt size constructor (it has the same name, but it's in a separate file):

datatypes/src/ParamShirtSizes.re

```
type shirtSize =
  | Small
  | Medium
  | Large
  | XLarge(int);
```

The last line says that to construct an XLarge variant, you need to provide an integer, which we'll use to tell how many "extras" we want:

datatypes/src/ParamShirtSizes.re

```
let mySize: shirtSize = Medium;
let bigSize = XLarge(1);
let veryBigSize = XLarge(3);
```

When it comes to setting the price, let's say that XLarge(1) costs $16.00, plus $0.50 for every additional X. We modify the switch to accept the parameter and use it:

datatypes/src/ParamShirtSizes.re

```
let price = (size: shirtSize) : float => {
  switch (size) {
    | Small => 11.00
    | Medium => 12.50
    | Large => 14.00
    | XLarge(n) => 16.00 +.
        (float_of_int(n - 1) *. 0.50)
  }
};

Js.log(price(mySize)); /* output: 12.5 */
Js.log(price(bigSize)); /* output: 16 */
Js.log(price(veryBigSize)); /* output: 17 */
```

Line 6 uses *destructuring* to extract the parameter from the size variable. For example, if size were XLarge(3), n would have the value 3 in the calculation. In addition to extracting parameters, destructuring also lets you extract fields from a data structure. We'll see this come into play in later chapters.

The next function to modify is stringOfShirtSize(). Again, we need destructuring to extract the parameter n in variants of the form XLarge(n) and make a string of that many Xs. The make() function in BuckleScript's String module[1] does exactly that.

1. reasonml.github.io/api/String.html

datatypes/src/ParamShirtSizes.re

```
let stringOfShirtSize = (size: shirtSize) : string => {
  switch (size) {
    | Small => "S"
    | Medium => "M"
    | Large => "L"
    | XLarge(n) => String.make(n, 'X') ++ "L"
  };
};

Js.log(stringOfShirtSize(veryBigSize)); /* output: XXXL */
```

Note that the second argument to make() is a character in single quotes. We'll solve the problem of repeating a multi-character string when we discuss recursion on on page ?.

The shirtSizeOfString() function needs the addition of a few lines to handle the new "extra" sizes (showing only the additions here):

datatypes/src/ParamShirtSizes.re

```
| "L" => Large
| "XL" => XLarge(1)
| "XXL" => XLarge(2)
| "XXXL" => XLarge(3)
| "XXXXL" => XLarge(4)
| _ => Medium
```

This function still leaves the issue of what to do with invalid strings—it's still blindly assigning Medium. Let's find a better way to handle this after you first try your hand at creating a variant data type.

Using One-Variant Data Types

Now that we know how to create a variant data type with a parameter, we can improve on the bogus example that we made with aliases on page ?.

Instead of aliases, we define the score, percent, and user ID types as data type constructors:

datatypes/src/SingleVariant.re

```
type scoreType = Score(int);
type percentType = Percent(float);
type userId = UserId(int);
```

When we use variables of these parameterized types, we must construct values, as in line 1, and destructure them, as in line 4.

datatypes/src/SingleVariant.re

```
Line 1  let person: userId = UserId(60);
        -
        - let calcPercent = (score: scoreType, max: scoreType) : percentType => {
        -   let Score(s) = score;
      5 -   let Score(m) = max;
        -   Percent(float_of_int(s) /. float_of_int(m) *. 100.0);
        - };
        -
        - /* Won't compile. Comment out next line to get a working program */
     10 /*let result = calcPercent(person, Score(75));*//**/
        -
        - let Percent(result) = calcPercent(Score(40), Score(75));
        - Js.log({j|Good result is $result|j}); /* output: Good result is 53.33333... */
```

Using these data types gives us type safety. ReasonML will complain in line 10 that you're trying to use a `userId` where a `score` is required.