# Pattern Matching in TypeScript with Record and Type Patterns

Pattern matching allows programmers to compare data with defined structures to easily pick one of the available expressions. Many languages that are designed as 'functional programming languages' have build-in keywords for pattern matching. Well know examples are F# ( `match ... with` ) or Haskell ( `case ... of` ). One language that works very well with functional programming but lacks those features is TypeScript. In this article we will build a small libary to add support for pattern matching to TypeScript. We will implement a lot of advanced features like record, type and array patterns. We will also make sure that the type inference of TypeScript understands out patterns and narrows the types when ever possible.

## Using a Builder

The start of our library is a builder API that allows us to write fluent style code when adding patterns. We pass the `match` function the value that we whant to match. This returns a builder on which we can call `with` to add a pattern, can call `otherwise` to set the fallback value and can call `run` to find the matching pattern and execute its expression.

```
const match = <a, b>(value: a, otherwise: () => b, patterns: Array<[a, fun<a, b>]> = [
  with: (pattern: a, expr: fun<a, b>) =>
    match(value, otherwise, [...patterns, [pattern, expr]]),
  otherwise: (otherwise: () => b) => match(value, otherwise, patterns)
  run: (): b => {
    const p = patterns.find(p => match_pattern(value, p[0]))
    if (p == undefined) return otherwise()
    return p[1](value)
  }
})

const match_pattern = <a>(value: a, pattern: a) => a === a
```

This implementation is at te moment nothing more than a (complicated) lookup table, but we will be adding all the promised features futher down the line. An example of how to use our library:

```
match(1)
  .with(0,    v => '000')
  .with(1,    v => v * 2)
  .with(2,    v => v * v)
  .otherwise(() => 'nope')
  .run()
```

## Record Patterns

The first extension we will add are record patterns. Record patterns allow us to create patterns that describe the stucture of an object and will match when the input data matches the provided structure. A record pattern itself is an object where the values are the patterns for their respective keys. Nesting will be fully supported to allow the description of very complex objects. To implement this feature we will introduce a new type `Pattern<a>` that will describe the valid patterns for a given type `a` . For now the implementation is the same as the `Partial` type of the standard library, but we will also expand this type later on. At last we update the `match_pattern` function to support objects.

```
type Pattern<a> = { [ k in keyof a ]?: Pattern<a[k]> }

const match_pattern = <a>(value: a, pattern: Pattern<a>) => typeof(value) != 'object'
  Object.keys(pattern).every(k => pattern[k] == undefined ? false : match_pattern(valu
```

Using this feature will look like this:

```
let vector = { x: 1, y: 1 }

match(vector)
  .with({ x: 1, y: 1, z: 1 }, () => 'vector3')
  .with({ x: 2, y:1 },        () => 'vector2')
  .with({ x: 1 },             () => 'vector1')
  .otherwise(                 () => 'no match')
  .run()
```

> Record patterns are very usefull for parsing untyped data from external sources, especially when combined with type patterns. I will show some examples of this later on.

## Type Inference

One of the really good things about TypeScript is the flow typesystem that narrows types down whenever a condition rules out possible values. We would like this to also happen with our pattern matching as well. Let's define an Option monad to use as an example:

```
type Option<a> = { kind: 'none' } | { kind: 'some', value: a }

let val: Option<string> = { kind: 'some', value: 'hello' }

match(val)
  .with({ kind: 'some' }, o => o.value)
  .run()
```

This code is now giving a type error because the compiler has no way of figuring out that the pattern implices that the option is a Some and does contains a value. Luckily TypeScript comes with a powerfull typesystem that provides the tools we need to solve challenges like this. We will be using the `typeof` operator to get the exact type of the pattern that the user provided and use `Extract` to narrow down the type of `a` . The code looks as follows:

```
with: <p extends Pattern<a>>(
  pattern: p,
  expr: (a: Extract<a, typeof pattern>) => b
) => match(value, otherwise, [...patterns, [pattern, expr]])
```

We introduced a new generic type `p` that has to be a subset of `pattern<a>`. Now we can use this stricter subset of `p` to narrow down `a`. The `typeof` operator is used to force TypeScript to inference the type of the provided pattern, as we want the smallest possible subset and not the full `Pattern<a>` type. The entire quality of our type inference depends on how good we are in narrowing down `p`! With those changes our example with option will have the correct types infered as proven by this screenshot:

[Screenshot to proof it]

While this solution adds a lot of power and safety to our library when dealing with typed (discrimated) unions, there is one major problem with using the `Extract` type. If the input type of our pattern is `any` there will not be any type inference and the input type will be mapped to `any` (or `never` in some cases). This means that in its current state the library is not usable for parsing untyped data. The following screenshot highligths the problem:

[screenhot where any goes to any]

This behaviour can be explained when we look to the definition of `Extract`. It is defined as `a extends b ? a : never`. Because the type returns `a` (in our case `any`), we will always get `any` back when `any` goes in. The solution to this problem is to make a custom `extract` type that first checks if `b` extends `a` before it goes through the same logic as `Extract`. This means that when `a` is `any`, `b` (the pattern) will be returned.

```
type extract<a, b> = b extends a ? b : a extends b ? a : never
```

With this new `extract` type we have type inference that works on both typed and untyped data. Now we can match on exact values, branches of unions and on the structure of untyped data.

## Type Patterns

Type patterns are patterns that try to match the input value with a specific type. This is needed when we want to process data that is of an union type (like `string | number`) or is not typed at all (like a response from an API call). We will introduce a handfull of new patterns for this feature: `String`, `Number` and `Boolean` that will match their corresponding primitive types. We will also make sure that this feature properly intergrates with the type inference.

The first step is to update the type `Pattern<a>` to allow the patterns to be used. For example, if the input type is `number` we want to allow a match on both a `number` and `Number`. For this we use conditional types to map `a` to `Pattern<a>`:

```
type Pattern<a> =
  a extends number ? a | NumberConstructor :
  a extends string ? a | StringConstructor :
  a extends boolean ? a | BooleanConstructor :
  { [k in keyof a]?: Pattern<a[k]> }
```

This allows us to write a pattern like `{ Id: Number }` to match the id with any number. The next step is to implement the actual matching in the `match_pattern` function:

```
const match_pattern = <a>(value: a, pattern: Pattern<a>) => {
  if (pattern === String) return typeof(value) == 'string'
  if (pattern === Boolean) return typeof(value) == 'boolean'
  if (pattern === Number) return typeof(value) == 'number' && Number.isNaN(value) == fa
  if(typeof (value) != 'object') return value === pattern
  return Object.keys(pattern).every(k => pattern[k] == undefined ? false : match_patte
}
```

One last thing to do is undoing the modifications we made to `a` before we use the patterns to narrow down the input type. For this we need to map the contructor types to their instance types. In this way a match on `String` will give you a `string` in the result, like you would expect:

```
type InvertPattern<p> =
  p extends NumberConstructor ? number :
  p extends StringConstructor ? string :
  p extends BooleanConstructor ? boolean :
  { [k in keyof p]: InvertPattern<p[k]> }
```

And we update the definition of `with` to make use of this new type:

```
with: <p extends Pattern<a>>(
  pattern: p,
  expr: (a: extract<a, InvertPattern<typeof pattern>>) => b
) => match(value, otherwise, [...patterns, [pattern, expr]]),
```

With record and type patterns in place our library has grown to a powerfull tool for dealing with parsing of data. Let's imagine an external service that gives you either an object with an `errorMessage` or a Blog entry. We can simply parse this data with the following patterns while still validing every property of our response (in contrast with just casting the response to `Blog` with `as` like many people do):

```
interface Blog { id: number, title: string }

let httpResult: any = /* ... */

match<any, Blog | Error>(httpResult)
  .with({ Id: Number, Title: String }, r => ({ id: r.Id, title: r.Title }))
  .with({ errorMessage: String },      r => new Error(r.errorMessage))
  .otherwise(                          () => new Error('client parse error'))
  .run()
```

## Array Patterns

The last feature we will be adding are array patterns. Array patterns allow us to check if all items in an array match a pattern. The pattern for an array will be defined as an array with 1 item: the pattern that will be used for all items.

We will again start by updating our types and update the implementation later on. To create the type for the array pattern we will use the `infer` keyword to get the inner type of the array. With this type we will define a singleton array with a pattern for the inner type. The `InvertPattern` type will use the same features to revert this process and make sure we still infer the correct types in the expression.

```
type Pattern<a> =
  a extends number ? a | NumberConstructor :
  a extends string ? a | StringConstructor :
  a extends boolean ? a | BooleanConstructor :
  a extends Array<infer aa> ? [Pattern<aa>] :
  { [k in keyof a]?: Pattern<a[k]> }

type InvertPattern<p> =
  p extends NumberConstructor ? number :
  p extends StringConstructor ? string :
  p extends BooleanConstructor ? boolean :
  p extends Array<infer pp> ? InvertPattern<pp>[] :
  { [k in keyof p]: InvertPattern<p[k]> }
```

> Note: this uses circular type definitions that are only supported in the latetst releases of TypeScript.

Finally we update the `match_pattern` function to check if the pattern is an array and execute the matching:

```
const match_pattern = <a>(value: a, pattern: Pattern<a>) => {
  /* String, Number, Boolean patterns */
  if (Array.isArray(pattern)) {
    if (!Array.isArray(value)) return false
    return value.every(v => match_pattern(v, pattern[0]))
  }
  if (typeof (value) != 'object') return value === pattern
  return Object.keys(pattern).every(k => pattern[k] == undefined ? false : match_patte
}
```

A very practical usecase for this feature is parsing the data for an overview:

```
let blogOverviewResponse: any = /* ... */

match<any, Blog[] | Error>(blogOverviewResponse)
  .with([{Id: Number, Title: String}], r => r.map(b => ({id: b.Id, title: b.Title})))
  .with({ errorMessage: String },      r => new Error(r.errorMessage))
  .otherwise(                         () => new Error('client parse error'))
  .run()
```

## Conclusion

In this article we have seen how we can use TypeScript's extensive typesystem to add complety new features to the language. We have also seen that pattern matching allows us to write code that is more declarative than writing a lot of conditions. This more abstract way of thinking allows us to describe complex parsing logic with simple, short and typesafe patterns. The complete source code can be found here. This is somewhat more extensive version than is show here, but I couldn't include everything in this article without making it way to long.