Code branching in Typescript

How to if/else

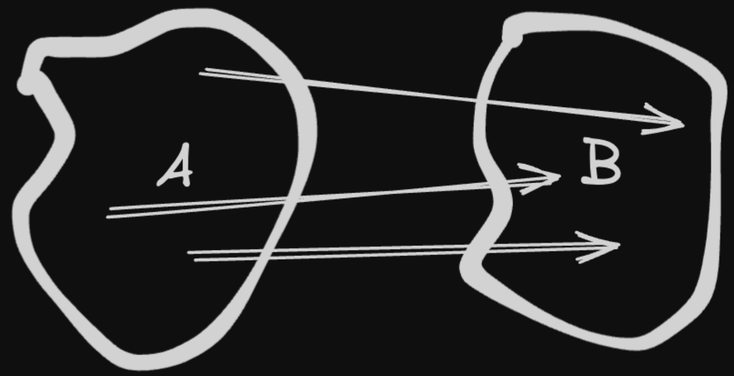
about:author

- https://github.com/dearlordylord
- Works at Monadical https://monadical.com/
- https://www.loskutoff.com/blog
- Types are strong with this one

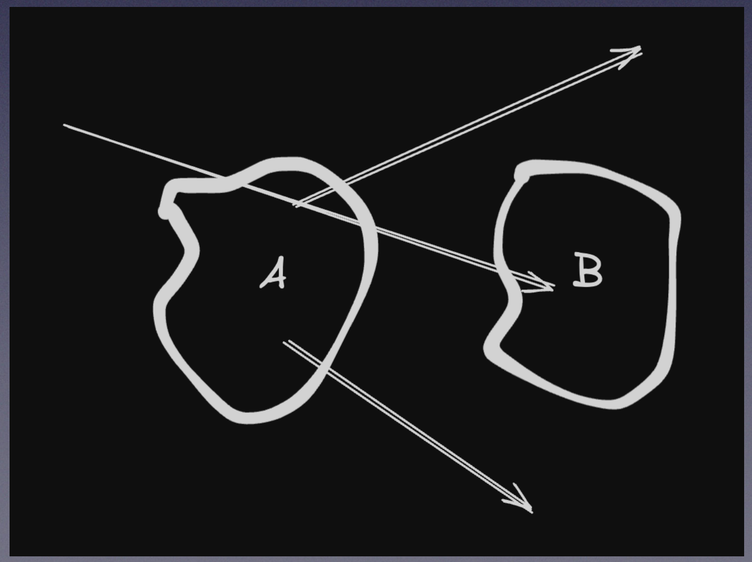
tldr,

```
if (a) {
  doThis()
} else {
  doThat()
}
```

vs.



and how to not…



Branching is everywhere

- Redux / useReducer
- Event sourcing / **C**QRS
- Webhooks (Stripe etc)
- Queues (Kafka topics etc)
- Any custom if/else, switch/case, dictionaries

Example

```
type State = {
  number: number;
}

type Action = {type: 'increment'} | {type: 'decrement'};

const reduce = (state: State, action: Action): State => {
  switch (action.type) {
    case 'increment':
      return {...state, number: state.number + 1};
    default:
      throw new Error('Unknown action type');
  }
};
```

“Entanglements”

- Side effects
- Dependencies (dependency injection)
- Error handling (error values, exceptions)
- Async
- Transactions

With this context, branching becomes tricky.
A primitive if/else is difficult.

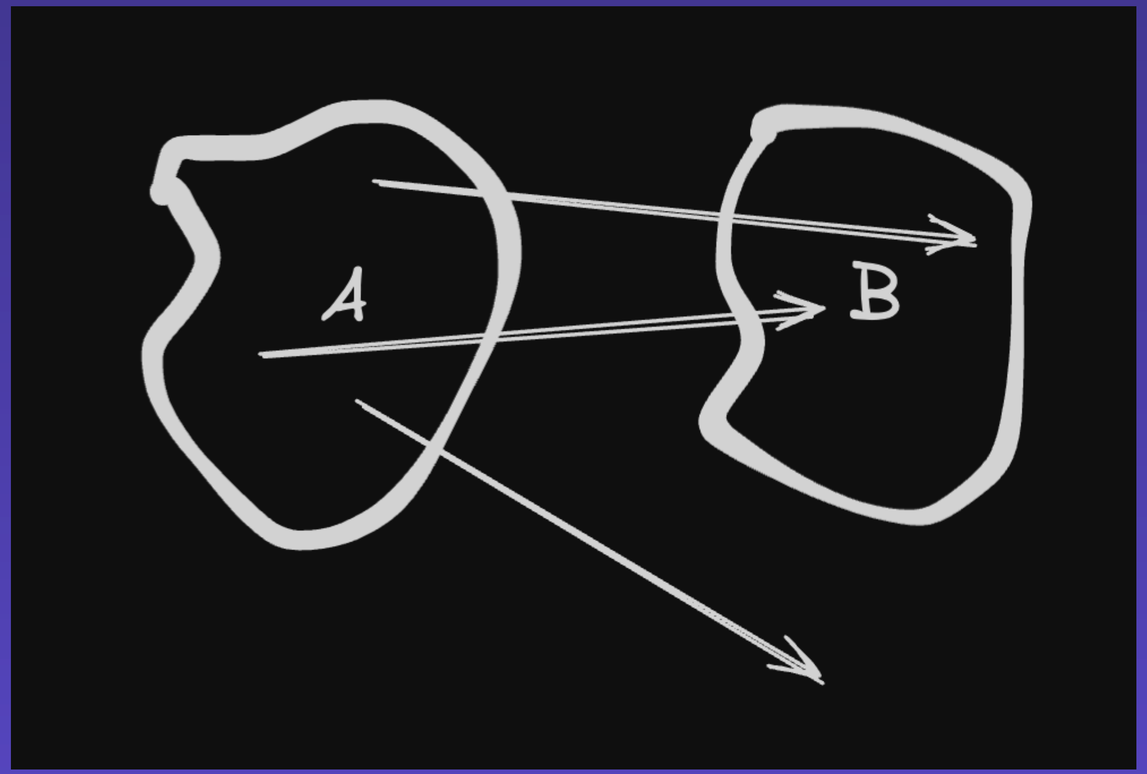
Example poisoned

```
const reducer = async (state: State, action: Action): Promise<State> => {
  switch (action.type) {
    case 'increment': {
      // we tie Async to promise API (there are other async implementations! Task of fp-ts/fp-tx,
Effect)
      const n = await readDb(); // also error leaks here
      // and where readDb() comes from?
      const newNumber = state.number + n;
      if (newNumber > MAX_NUMBER) throw new Error('Overflow!');
      return {...state, number: state.number + n};
    }
    default:
      throw new Error('Unknown action type');
  }
};
```

Thus, let’s get branching in order before we have to mix it with complex concepts

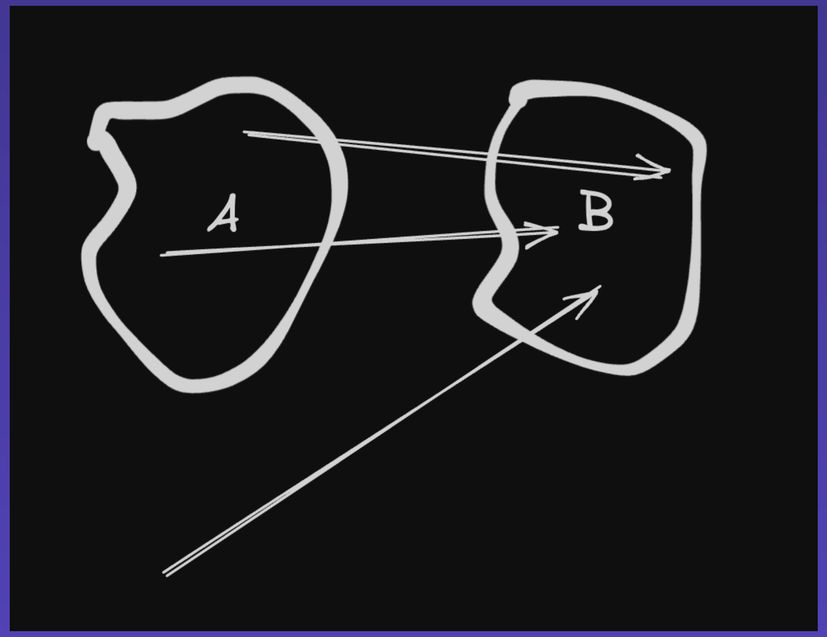
Exhaustiveness

- https://github.com/dearlordylord/branching/index.ts



Matching and type narrowing

- https://github.com/dearlordylord/branching/index.ts



(approximate intuition ^)

Pattern matching

- More than one discriminator?
- Combinatoric cases, runtime checks?
- https://tc39.github.io/proposal-pattern-matching
- https://github.com/gvergnaud/ts-pattern

```ts
import { match, P } from 'ts-pattern';

type Data =
  | { type: 'text'; content: string }
  | { type: 'img'; src: string };

type Result =
  | { type: 'ok'; data: Data }
  | { type: 'error'; error: Error };

const result: Result = ...;

const html = match(result)
  .with({ type: 'error' }, () => <p>Oups! An error occured</p>)
  .with({ type: 'ok', data: { type: 'text' } }, (res) => <p>{res.data.content}</p>)
  .with({ type: 'ok', data: { type: 'img', src: P.select() } }, (src) => <img src={src} />)
  .exhaustive();
```

Also, returns a value like:

```rust
let result = if condition { value1 } else { value2 };
```

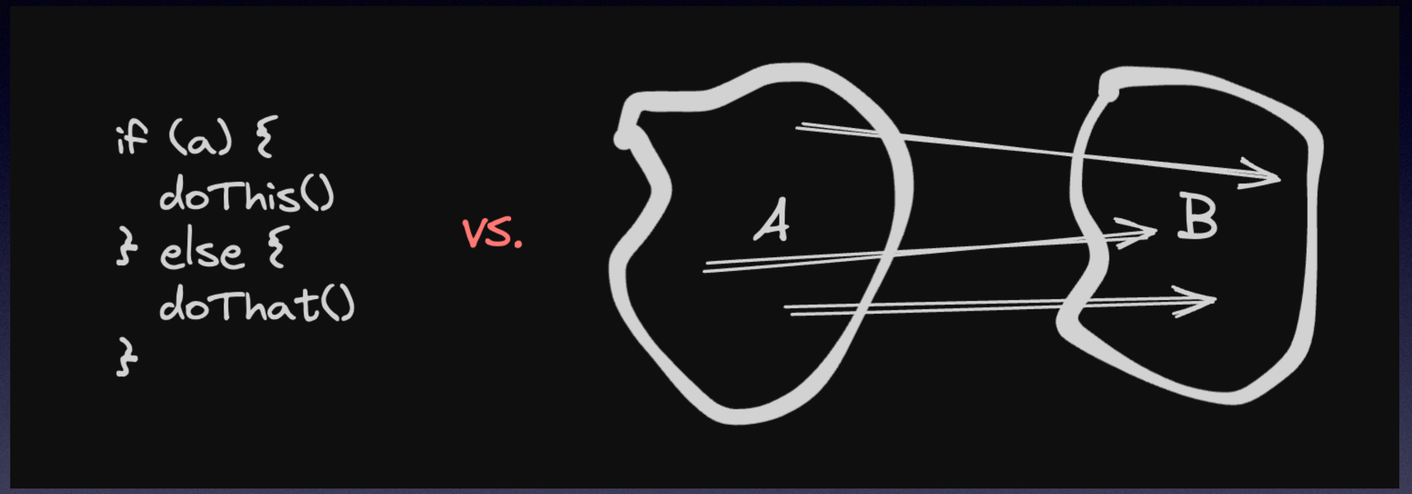
```scala
val minValue = if (a < b) a else b
```

… etc

Ts-pattern

- Good for complex matching that basic tooling like if/else, switch/case, inheritance, object mapping can’t do
- Performance overhead (builder pattern inside)

Thanks you!



Questions

- e.g. “where’s if/else…?”

Bonus: visitor pattern and entity/behaviour combinatorics

- https://github.com/dearlordylord/branching/visitor.ts