From Promises to Power

Enhancing Web Reliability

Gabriel Bornea | ING Hubs Romania

about me

- Gabriel Bornea (@gabi_b28 / gabriel-bornea)
- Chapter Lead / Software Engineer @ ING Hubs Romania
- **Languages**: Kotlin, TypeScript, Scala, Java, JavaScript
- **Interests**: Functional Programming

our path to resilience

Challenge:

- Starting a new frontend project with a focus on resilience.

Goals:

- Abstracting away async computations, retries, error handling.

our requirements

Defining our needs:

- Efficient handling of asynchronous operations.
- Robust retry mechanism for reliability.
- Comprehensive and clear error handling.

why typescript?

- Robust type system for catching errors early.
- Seamless integration with existing JavaScript code.
- Enhanced developer productivity and code maintainability.
- Strong community and extensive ecosystem.

our solution

Exiting libraries were either too complex or quite inflexible for our needs.

Build our own library:

- Abstract away asynchronous operations, retries and error handling.
- Streamlined codebase with a clean, familiar API.
- Rooted in functional programming principles for reliability and maintainability.

monadyssey

why functional programming?

Functional programming provides several benefits for building resilient and reliable systems.

- Better modularity (pure functions, higher-order functions).
- Easier reasoning (immutable values, referential transparency).
- Predictability (no side effects, deterministic behaviour).

what is functional programming?

In very simple terms, functional programming is a way of writing applications using only **pure functions** and **immutable values**.

why use immutable values?

- Data does not change once created, ensuring consistency across the system.
- Reduces unexpected behaviours and side effects.
- Easier to manage state in concurrent programming since data cannot be modified by multiple threads.

pure functions

Functions that always produce the **same output** for the **same input** and have **no side effects**.

A pure function is a computational analogue of a mathematical function.

side effects

Functions that produce some **observable effect** besides returning a value.

```
f(x: Int) -> { x + 1          if x >= 0
             { throw Error    if x < 0
```

Examples of **Side Effects**:

- Reading input from the user
- Making an HTTP request to fetch data from a server
- Logging information for debugging or auditing purposes
- Throwing an error in response to an invalid operation

partial and total functions

```
f : x -> { x + 1          if x >= 0
         { throw Error    if x < 0
```

```
f : x -> Int
```

Partial function

```
f : x -> Int | Error
```

Total function

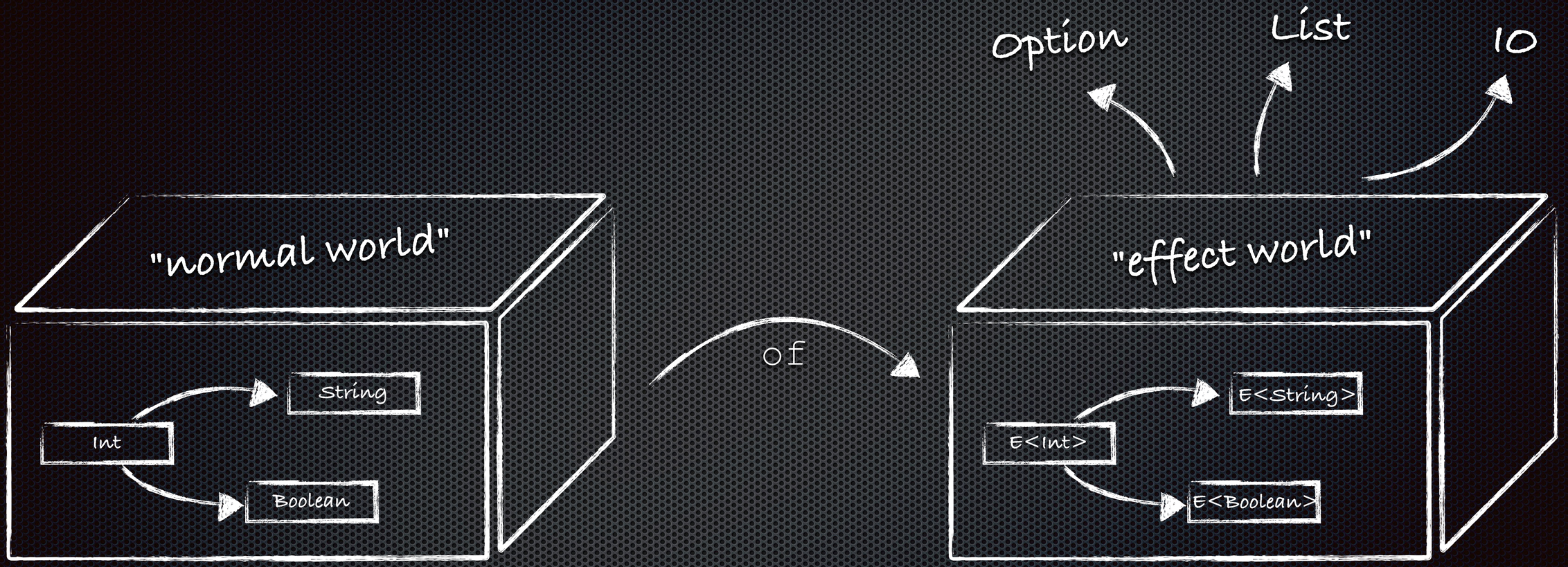
Total functions ensure all inputs are handled and possible return types are clear, leading to robust, predictable, and maintainable code.

pure functions, no side effects?

"Side effects are what we get paid to do every day." - Russ Olsen

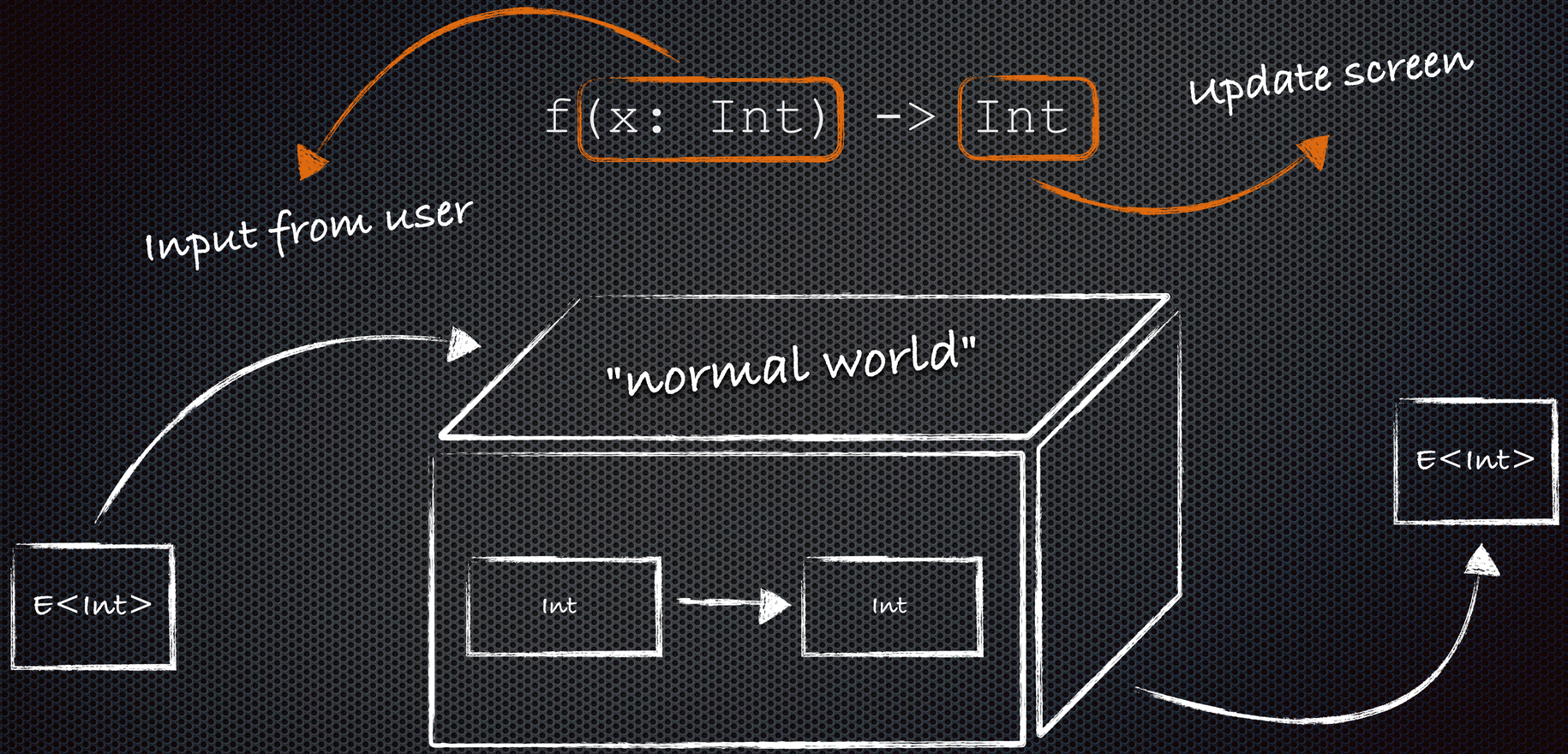
The whole purpose of functional programming isn't to prevent side-effects, it is just to push side-effects to the boundaries of your system in a well-known and controlled way.

understanding effects vs. side effects



In functional programming, effects are controlled and managed within the “effect world”, minimizing side effects in the “normal world”.

pushing side-effects to the boundaries



Business logic remains pure and free of side effects, with effects wrapping only the input and output operations.

From Concept to Implementation

- Retrieve the current geographic coordinates (latitude and longitude) based on the user's IP address.
- Using the obtained coordinates, fetch the current weather data.
- Transform the fetched weather data into a user-friendly format.
- Present the formatted weather information to the user on the screen.

managing side effects

unrefined implementation without IO

```
async getCurrentWeather(): Promise<void> {
  const locationRes = await fetch('https://ipinfo.io/json');
  if (!locationRes.ok) {
    throw new Error("Failed to retrieve user location");
  }
  const location: CurrentLocation = await locationRes.json();

  const [latitude, longitude] = location.loc.split(",").map(Number);

  const currentWeatherRes = await fetch(`https://api.open-meteo.com/v1/forecast?latitude=${latitude}
&longitude=${longitude}&current_weather=true`);
  if (!currentWeatherRes.ok) {
    throw new Error("Failed to retrieve current weather conditions");
  }

  const weather: Weather = await currentWeatherRes.json();

  this.conditions = {
    // ...
  }
}
```

difficult to test

no error recovery

mixing business logic and side effects & direct state mutation

refactoring for manageability

breaking down monolithic code

```
async _getCurrentWeather(): Promise<void> {
  const locationRes = await fetch('https://ipinfo.io/json');
  if (!locationRes.ok) {
    throw new Error("Failed to retrieve user location");
  }
  const location: CurrentLocation = await locationRes.json();
  // ...
}
```

```
async getCurrentLocation(): Promise<CurrentLocation> {
  try {
    const response = await fetch('https://ipinfo.io/json');
    if (!response.ok) {
      return Promise.reject(new Error("Failed to retrieve user location"));
    }
    return await response.json();
  } catch (e: unknown) {
    const message = e instanceof Error ? e.message : "Failed to retrieve user location";
    return Promise.reject(new Error(message));
  }
}
```

refactoring for manageability

breaking down monolithic code

```
async getCurrentLocation(): Promise<CurrentLocation> {
  try {
    const response = await fetch('https://ipinfo.io/json');
    if (!response.ok) {
      return Promise.reject(new Error("Failed to retrieve user location"));
    }
    return await response.json();
  } catch (e: unknown) {
    const message = e instanceof Error ? e.message : "Failed to retrieve user location";
    return Promise.reject(new Error(message));
  }
}
```

```
export namespace HttpClient {
  export const request = async <A>(url: string): Promise<A> => {
    // ...
  };
}
```

refactoring for manageability

breaking down monolithic code

```
async getCurrentWeather(): Promise<void> {
  // ...
  const [latitude, longitude] = location.loc.split(“,”).map(Number);
  // ...
}
```

```
{
  "ip": "",
  "hostname": "",
  "city": "",
  "region": "",
  "country": "",
  "loc": "44.5500,26.0667",
  "org": "",
  "postal": "",
  "timezone": "",
  "readme": ""
}
```

```
getLatitudeAndLongitude(location: CurrentLocation): Promise<[number, number]> {
  if (!location.loc) {
    return Promise.reject(new Error("Location data is missing"));
  }
  const [latitude, longitude] = location.loc.split(",").map(Number);

  if (isNaN(latitude) || isNaN(longitude)) {
    return Promise.reject(new Error("Invalid latitude or longitude values"));
  }

  return Promise.resolve([latitude, longitude]);
}
```

refactoring for manageability

breaking down monolithic code

```
async getCurrentWeather(): Promise<void> {
  // ...
  const currentWeatherRes = await fetch(`https://api.open-meteo.com/v1/...`);
  if (!currentWeatherRes.ok) {
    throw new Error("Failed to retrieve current weather conditions");
  }
  const weather: Weather = await currentWeatherRes.json();
  // ...
}
```

```
const getCurrentWeatherData = async (latitude: Number, longitude: Number): Promise<Weather> =>
  HttpClient.request(`https://api.open-meteo.com/v1/...`)
```

refactoring for manageability

breaking down monolithic code

```
async getCurrentWeather(): Promise<void> {
  // ...
  this.conditions = {
    city: location.city,
    country: location.country,
    temperature: weather.current_weather.temperature,
    temperatureUnit: weather.current_weather_units.temperature,
    windSpeed: weather.current_weather.windspeed,
    windSpeedUnit: weather.current_weather_units.windspeed,
    windDirection: weather.current_weather.winddirection,
    windDirectionUnit: weather.current_weather_units.winddirection
  }
}
```

```
mapToConditions(location: CurrentLocation, weather: Weather): CurrentConditions {
  return {
    city: location.city,
    // ...
  }
}
```

refactoring for manageability

breaking down monolithic code

```
async getCurrentWeather(): Promise<CurrentConditions> {
  const location = await this.getCurrentLocation();
  const [latitude, longitude] = await this.getLatitudeAndLongitude(location);
  const weather = await this.getCurrentWeatherData(latitude, longitude);
  return this.mapToConditions(location, weather);
}
```

Some benefits of refactoring:

- Easier to test: Everything can be tested in isolation.
- Separation of concerns: IO operations are decoupled from business logic.
- Improved error handling: Errors are not thrown but managed within promises.

That's it, right?

Not really, let's dive deeper into how the IO data type can further enhance our approach.

introducing the IO data type

managing side effects in a controlled way

The `IO` data type is a core component in *monadyssey* for handling side effects in a controlled and composable manner.

understanding the IO data type

managing side effects in a controlled way

- The IO data type represents a computation that, when executed, performs side effects and yields a value.
- It represents side effects as values, making them easier to manage and compose.
- IO wraps an impure computation, delaying its execution until explicitly called. This allows side effects to be controlled and isolated.

enhancing asynchronous operations

transitioning from Promises to IO for better control

```
                                  ApplicationError
const getCurrentLocation = (): IO<Error, CurrentLocation> =>
  IO.of(
    async () => await HttpClient.request('https://ipinfo.io/json'),
    (e) => new UserLocationError(e instanceof Error ? e.message : "Failed to retrieve user location")
)
.handleErrorWith((e) => new UserLocationError(e instanceof Error ? e.message : "Failed to retrieve user location"));


class UserLocationError {
  message: string;
  constructor(message: string) {
    this.message = message;
  }
}

type ApplicationError = UserLocationError | WeatherRetrievalError;
```

Errors can be composed

An `ApplicationError` could be a `UserLocationError` or `WeatherRetrievalError` or ...

enhancing asynchronous operations

transitioning from Promises to IO for better control

```
const getLatitudeAndLongitude = (location: CurrentLocation): IO<ApplicationError, [number, number]> =>
  IO.ofSync(
    () => location.loc.split(",").map(Number) as [number, number],
    (e) => new InvalidLocationError(e instanceof Error ? e.message : "Failed to parse user location")
  ).refine(
    ([lat, lon]) => !isNaN(lat) && !isNaN(lon),
    () => new InvalidLocationError("Invalid latitude or longitude values")
  )
```

Ensures the values are valid

enhancing asynchronous operations

transitioning from Promises to IO for better control

```
const getCurrentWeatherData = (latitude: Number, longitude: Number): IO<ApplicationError, Weather> =>
  IO.of(
    async () => await HttpClient.request(`https://api.open-meteo.com/v1/...`)),
    (e) => new WeatherRetrievalError(e instanceof Error ? e.message : "Failed to retrieve ...“)
  );
```

```
const mapToConditions = (location: CurrentLocation, weather: Weather): CurrentConditions => {
  return {
    city: location.city,
    country: location.country,
    temperature: weather.current_weather.temperature,
    temperatureUnit: weather.current_weather_units.temperature,
    windSpeed: weather.current_weather.windspeed,
    windSpeedUnit: weather.current_weather_units.windspeed,
    windDirection: weather.current_weather.winddirection,
    windDirectionUnit: weather.current_weather_units.winddirection
  }
}
```

enhancing asynchronous operations

transitioning from Promises to IO for better control

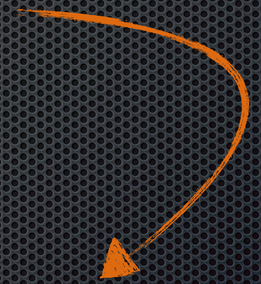
```
getCurrentWeather(): IO<ApplicationError, CurrentConditions> {
  return getCurrentLocation()
    .flatMap((location) =>
      getLatitudeAndLongitude(location)
        .flatMap(([lat, lon]) =>
          getCurrentWeatherData(lat, lon)
            .map((weather) =>
              mapToConditions(location, weather)
            )
        )
    );
}
```

- Centralized and consistent error handling within the IO context.
- Reusability and maintainability.
- Immutability and composability.

enhancing asynchronous operations

transitioning from Promises to IO for better control

```
async getCurrentWeather(): Promise<CurrentConditions> {
  const location = await getCurrentLocation();
  const [latitude, longitude] = await getLatitudeAndLongitude(location);
  const weather = await getCurrentWeatherData(latitude, longitude);
  return mapToConditions(location, weather);
}
```



```
getCurrentLocation()
  .flatMap((location) => getLatitudeAndLongitude(location).flatMap(([lat, lon]) =>
    getCurrentWeatherData(lat, lon).map((weather) => mapToConditions(location, weather))));
```

- The IO-based approach loses the straightforward, step-by-step readability of async/await.
- Requires understanding and managing nested function calls, which can impact readability and make the code harder to follow.

enhancing asynchronous operations

transitioning from Promises to IO for better control


```
getCurrentWeather = (): IO<ApplicationError, CurrentConditions> =>
  IO.forM(async bind => {
    const location = await bind(getCurrentLocation());
    const [latitude, longitude] = await bind(getLatitudeAndLongitude(location));
    const weather = await bind(getCurrentWeatherData(latitude, longitude));

    return mapToConditions(location, weather);
  });
```

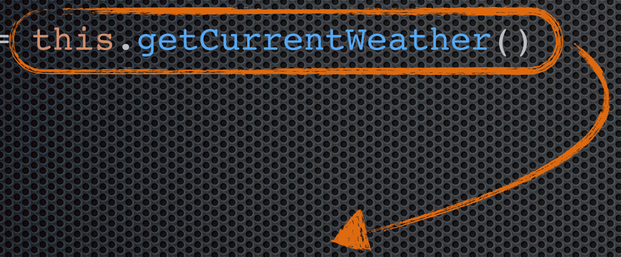
enhancing asynchronous operations

transitioning from Promises to IO for better control

```
getCurrentWeather = (): IO<ApplicationError, CurrentConditions> =>
  IO.forM(async bind => {
    const location = await bind(getCurrentLocation());
    const [latitude, longitude] = await bind(getLatitudeAndLongitude(location));
    const weather = await bind(getCurrentWeatherData(latitude, longitude));

    return mapToConditions(location, weather);
  })

const conditions = this.getCurrentWeather()
```



IO operations are not executed until explicitly run

- *Deferred Execution*: Control when side effects occur.
- *Composability*: Easily combine and transform IO operations without triggering them.

enhancing asynchronous operations

transitioning from Promises to IO for better control

```
getCurrentWeather = (): IO<ApplicationError, CurrentConditions> =>
  IO.forM(async bind => {
    const location = await bind(getCurrentLocation());
    const [latitude, longitude] = await bind(getLatitudeAndLongitude(location));
    const weather = await bind(getCurrentWeatherData(latitude, longitude));

    return mapToConditions(location, weather);
  })


    Err<ApplicationError> | Ok<CurrentConditions>
const result = await this.getCurrentWeather().runAsync();
```

- *runAsync* returns an Err or Ok type, which ensures that any errors are explicitly handled, enhancing code reliability and robustness.
- By returning a union type (Err | Ok), the code becomes more type-safe, making it easier to understand and maintain.

enhancing asynchronous operations

transitioning from Promises to IO for better control

```
getCurrentWeather = (): IO<ApplicationError, CurrentConditions> =>
  IO.forM(async bind => {
    const location = await bind(getCurrentLocation());
    const [latitude, longitude] = await bind(getLatitudeAndLongitude(location));
    const weather = await bind(getCurrentWeatherData(latitude, longitude));

    return mapToConditions(location, weather);
  })
```

Handle unsuccessful path

```
await this.getCurrentWeather().fold(
  (e: ApplicationError) => console.error(e.message),
  (conditions: CurrentConditions) => this.conditions = conditions
);
```

Handle successful path

What about retries?

Applications fail, it's just a matter of when, not if.

introducing the Schedule data type

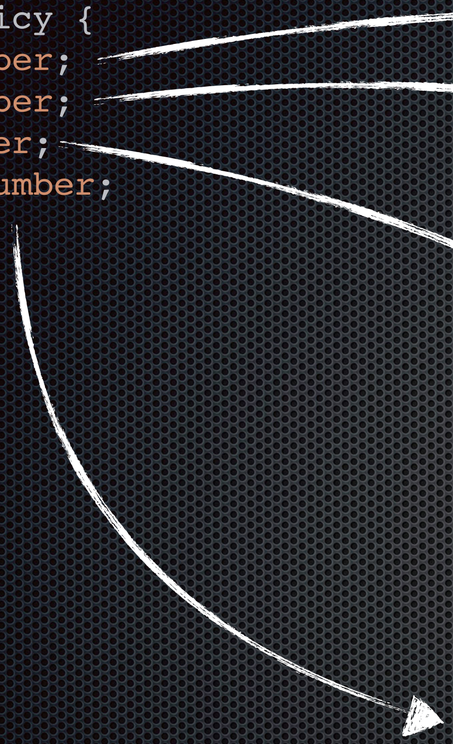
retrying on failures

monadyssey offers a Schedule type, coupled with a Policy configuration, enabling the declarative management of retries, delays, and timeouts. This approach enhances the resilience of operations by systematically handling failures.

introducing the Schedule data type

retrying on failures

```
interface Policy {
  recurs: number;
  factor: number;
  delay: number;
  timeout?: number;
}
```



maximum number of retry attempts

factor by which the delay increases after each retry

initial delay in milliseconds before the first retry

maximum duration in milliseconds that each attempt can take before timing out, if not set, attempts will not time out

introducing the Schedule data type

retrying on failures

```
const policy: Policy = {
  recurs: 3,
  factor: 1.2,
  delay: 1000,
  timeout: 3000
};

const scheduler = new Schedule(policy);

retryIf<E, A>(eff: IO<E, A>, condition: (error: E) => boolean): IO<E, A>

liftE: (error: Error) => E
```

the operation to retry if it fails

converts a general error into a specific error type

determines whether a failed attempt should be retried

introducing the Schedule data type

retrying on failures

```
type ApplicationError = UserLocationError | WeatherRetrievalError | ...;

class GenericError {
  constructor(public readonly message: string, public readonly retryable: boolean) {}
}

class UserLocationError extends GenericError {
  constructor(message: string) {
    super(message, true);
  }
}
```

retry only where it makes sense

introducing the Schedule data type

retrying on failures

```
scheduler
  .retryIf(
    this.getCurrentWeather(),
    (e: ApplicationError) => e.retryable,
    (e: Error) => new WeatherRetrievalError(e.message)
  ).fold(
     (e: ApplicationError) => console.error(e.message),
     (conditions: CurrentConditions) => this.conditions = conditions
   );
```

parallel operations

leveraging `parZip` for efficient user data validation

Scenario: Register a new user and validate different aspects of user data concurrently. Moreover, collect all possible errors at once.

- **Validate email:** Check if the email is properly formatted and not already in use.
- **Validate username:** Ensure the username is unique and follows specific guidelines.
- **Validate Password:** Verify that the password meets security standards.

Note: Achieving this level of parallel validation and error collection can be quite challenging using only promises and plain JavaScript or TypeScript. `parZip` simplifies this process, making the code more manageable and robust.

parallel operations

leveraging `parZip` for efficient user data validation

```
interface User {
  email?: string,
  username?: string,
  password?: string,
}


type ValidationResult<A> = { type: "Ok"; value: A } | { type: "Err"; error: string };

const validateEmail = async (email: string): Promise<ValidationResult<string>>
const validateUsername = async (username: string): Promise<ValidationResult<string>>
const validatePassword = async (password: string): Promise<ValidationResult<string>>

const validateUser = async (email: string, username: string, password: string): Promise<User> => {
  const [emailRes, usernameRes, passwordRes] = await Promise.all([
    validateEmail(email),
    validateUsername(username),
    validatePassword(password)
  ]);
  // . . .
}
```

parallel operations

leveraging `parZip` for efficient user data validation

```
const validateUser = async (email: string, username: string, password: string): Promise<User> => {
  const [emailRes, usernameRes, passwordRes] = await Promise.all([
    validateEmail(email),
    validateUsername(username),
    validatePassword(password)
  ]);
  let user: User = {};
  const errors: string[] = [];
  if (emailRes.type === "Err") {
    errors.push(emailRes.error);
  } else {
    user.email = emailRes.value;
  }
  // ...
  if (errors.length > 0) {
    return Promise.reject(errors);
  } else {
    return user;
  }
}
```

Code duplication

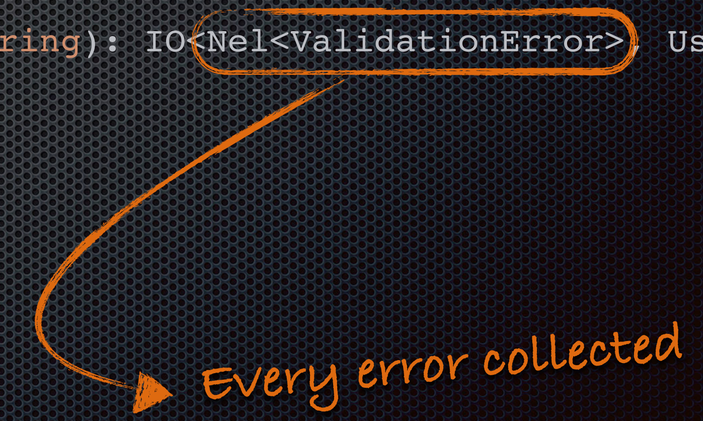
parallel operations

leveraging `parZip` for efficient user data validation

```
type ValidationError = InvalidUsernameError | InvalidEmailError | InvalidPasswordError;

const validateUsername = (username: string): IO<ValidationError, string>
const validateEmail = (email: string): IO<ValidationError, string>
const validatePassword = (password: string): IO<ValidationError, string>

const validateUser = (email: string, username: string, password: string): IO<Nel<ValidationError>, User> =>
  IO.parZip(
    validateUsername(username),
    validateEmail(email),
    validatePassword(password),
    (username, email, password) => {
      return {
        username: username,
        email: email,
        password: password
      }
    }
  )
```



benefits of using the IO data type

- **Centralized Error Management**: Unified handling within the IO context, reducing scattered try-catch blocks.
- **Declarative Asynchronous Control**: Define and manage async operations declaratively, enhancing readability and maintainability.
- **Decoupling and Lazy Evaluation**: Leads to cleaner, modular code and prevents unintended side effects by executing IO operations only when needed.
- **Functional Composition and Immutability**: Build complex operations from simple, reusable IO actions, ensuring predictability and reliability.
- **Automatic Retries and Resilience**: Built-in support for retrying operations on failure, enhancing resilience with configurable policies.
- **Explicit Side Effect Management**: Enhance clarity by making side effects explicit, leading to safer programs.
- **Enhanced Testability**: Easily mock or replace IO operations in tests, improving testability and allowing isolated testing of business logic.

other *monadyssey* data types

```
Option.ofNullable(42);
```

Option<A>

Represents a value that may or may not be present. It is a safer alternative to using `null` or `undefined`. An `Option` can either be some value (`Some<A>`) or none (`None`), allowing you to handle the presence or absence of a value explicitly.

other *monadyssey* data types

```
Either.catch(() => 42);
```

Either<E, A>

Represent a value that can be one of two possible types. It is typically used for error handling, where `Either` can be a value of type `A` (representing success) or of type `E` (representing failure). This makes it easier to manage and propagate errors in a type-safe manner.

other *monadyssey* data types

```
NonEmptyList.fromArray([42]);
```

NonEmptyList<A>

A list that is guaranteed to have at least one element. It ensures that operations that require a non-empty collection can be performed safely without additional checks for emptiness. This type is useful for operations where an empty list would be meaningless or result in an error.

other *monadyssey* data types

```
Eval.lazy(() => 42);
```

`Eval<A>`

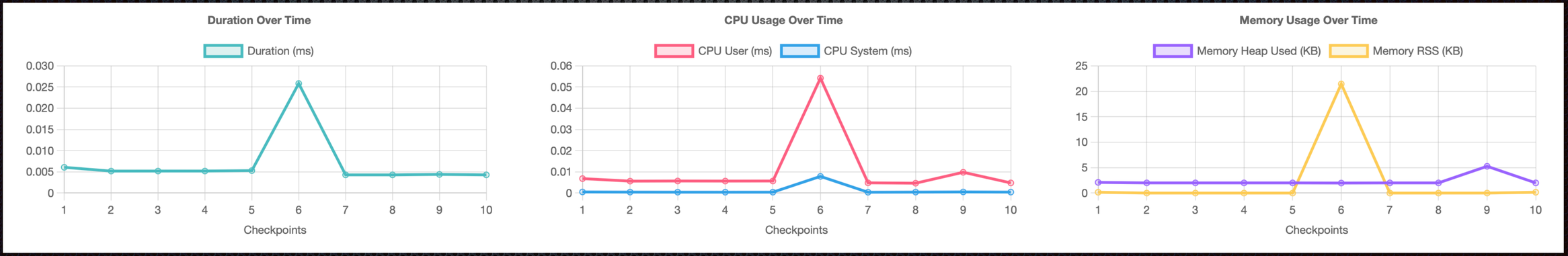
Controls evaluation strategies, such as lazy evaluation. It allows computations to be deferred and memoized, enabling more efficient and predictable performance. This is particularly useful for handling large computations or recursive algorithms.

performance considerations

monadyssey introduces some performance overhead but provides significant benefits in reliability and maintainability.

`IO.of()`

1000 repetitions



performance considerations

Performance Overhead

- For most applications, the benefits outweigh the performance costs.
- In highly performance-critical systems, the overhead might be noticeable.

Performance Metrics

- **Duration:** Slight increase in execution time due to added abstractions.
- **CPU Usage:** Higher CPU usage, particularly during intensive IO operations.
- **Memory Usage:** Increased memory footprint due to immutability and additional data structures.

When to Use

- Ideal for projects where reliability, maintainability, and modularity are priorities.
- For performance-critical applications, a careful evaluation is recommended.

Special Thanks

inspired by

https://www.scala-lang.org/

https://flix.dev/

https://arrow-kt.io/

https://typelevel.org/cats-effect/

https://zio.dev/

useful links

https://github.com/gabriel-bornea/monadyssey

https://github.com/gabriel-bornea/monadyssey-showcase

https://discord.gg/8gHQBRdxup

Thank you

Q&A