Namespace Fl.Functional.Utils

Classes

Functional

Class Functional

Namespace: Fl.Functional.Utils

Assembly: Fl.Functional.Utils.dll

```
public static class Functional
```

Inheritance

object ← Functional

Inherited Members

object.Equals(object) , object.Equals(object, object) , object.GetHashCode() , object.GetType() , object.MemberwiseClone() , object.ReferenceEquals(object, object) , object.ToString()

Methods

BindLeftAsync<TL, TR, TMl>(Task<Either<TL, TR>>, Func<TL, Either<TMl, TR>>)

Awaits `this` and applies `onLeft` to the `Left` value, allowing the error type to be transformed while keeping any `Right` value unchanged.

```
public static Task<Either<TMl, TR>> BindLeftAsync<TL, TR, TMl>(this Task<Either<TL,
TR>> @this, Func<TL, Either<TMl, TR>> onLeft)
```

Parameters

`this` Task<Either<TL, TR>>

A task that resolves to an LanguageExt.Either<L, R>.

`onLeft` Func<TL, Either<TMl, TR>>

A function that maps the left value to a new LanguageExt.Either<L, R>.

Returns

Task<Either<TMl, TR>>

A task that resolves to the original `Right` value unchanged, or to the LanguageExt.Either<L, R> produced by `onLeft` when the source is `Left`.

Type Parameters

`TL`

The original left (error) type.

`TR`

The right (success) type, unchanged by this operation.

`TMl`

The new left type produced by `onLeft`.

Combine<T>(Action<T>, Action<T>)

Merges two Action<T> delegates into one. When `nullableAction` is `null`, only `defaultAction` is returned; otherwise both actions are composed and executed in sequence.

```
public static Action<T> Combine<T>(this Action<T> nullableAction,
Action<T> defaultAction)
```

Parameters

`nullableAction` Action<T>

The primary action, which may be `null`.

`defaultAction` Action<T>

The fallback action used when `nullableAction` is `null`, and appended after it when it is not.

Returns

Action<T>

A combined Action<T> that runs `nullableAction` followed by `defaultAction`, or just `defaultAction` when the former is `null`.

Type Parameters

`T`

The type of the argument accepted by both actions.

DoAsync<T>(T, Func<T, Task>)

Asynchronously invokes `action` with `this` purely for its side effect. Returns the resulting Task so the caller can `await` completion.

```
public static Task DoAsync<T>(this T @this, Func<T, Task> action)
```

Parameters

`this` T

The value to pass to `action`.

`action` Func<T, Task>

The asynchronous side-effecting function to execute.

Returns

Task

A Task representing the asynchronous operation.

Type Parameters

`T`

The type of the value.

Do<T>(T, Action<T>)

Invokes `action` with `this` purely for its side effect. Unlike `Tee`, this method returns `void` and does not pass the value further.

```
public static void Do<T>(this T @this, Action<T> action)
```

Parameters

`this` T

The value to pass to `action`.

`action` Action<T>

The side-effecting action to execute.

Type Parameters

`T`

The type of the value.

ForEach<T>(IEnumerable<T>, Action<T>)

Iterates over `items` and invokes `action` for each element. Returns `None` when `items` is `null`, so callers can safely handle the missing-collection case via `Option` pattern matching.

```
public static Option<Unit> ForEach<T>(this IEnumerable<T> items, Action<T> action)
```

Parameters

`items` IEnumerable<T>

The collection to iterate. May be `null`.

`action` Action<T>

The action to invoke for each element.

Returns

Option<Unit>

`Some(LanguageExt.Unit)` after the iteration completes, or `None` when `items` is `null`.

Type Parameters

`T`

The element type of the collection.

MakeEitherAsync<TL, TR>(Task<TR>, Func<TL>)

Asynchronously wraps the awaited value in `Right`. Returns `Left` produced by invoking `leftFunc` only when the awaited value is `null`.

```
public static Task<Either<TL, TR>> MakeEitherAsync<TL, TR>(this Task<TR> @this,
Func<TL> leftFunc)
```

Parameters

`this` Task<TR>

A task whose result is wrapped in an LanguageExt.Either<L, R>.

`leftFunc` Func<TL>

A factory invoked to produce the `Left` value when the awaited value is `null`.

Returns

Task<Either<TL, TR>>

A task that resolves to `Right(value)` or `Left` from `leftFunc`.

Type Parameters

`TL`

The left (error) type.

`TR`

The right (success) type.

MakeEitherAsync<TL, TR>(Task<TR>, Predicate<TR>, Func<TL>)

Asynchronously wraps the awaited value in `Right` when `leftWhen` returns `false`; otherwise returns `Left` produced by invoking `leftFunc`.

```
public static Task<Either<TL, TR>> MakeEitherAsync<TL, TR>(this Task<TR> @this,
Predicate<TR> leftWhen, Func<TL> leftFunc)
```

Parameters

`this` Task<TR>

A task whose result is evaluated.

`leftWhen` Predicate<TR>

A predicate that, when satisfied, produces the `Left` case.

`leftFunc` Func<TL>

A factory invoked lazily to produce the `Left` value.

Returns

Task<Either<TL, TR>>

A task that resolves to `Right(value)` or `Left` from `leftFunc`.

Type Parameters

`TL`

The left (error) type.

`TR`

The right (success) type.

MakeEitherAsync<TL, TR>(Task<TR>, Predicate<TR>, TL)

Asynchronously wraps the awaited value in `Right` when `leftWhen` returns `false`; otherwise returns `Left(leftValue)`.

```
public static Task<Either<TL, TR>> MakeEitherAsync<TL, TR>(this Task<TR> @this,
Predicate<TR> leftWhen, TL leftValue)
```

Parameters

`this` Task<TR>

A task whose result is evaluated.

`leftWhen` Predicate<TR>

A predicate that, when satisfied, produces the `Left` case.

`leftValue` TL

The value used for `Left` when the predicate is `true`.

Returns

Task<Either<TL, TR>>

A task that resolves to `Right(value)` or `Left(leftValue)`.

Type Parameters

`TL`

The left (error) type.

`TR`

The right (success) type.

MakeEitherAsync<TL, TR>(Task<TR>, TL)

Asynchronously wraps the awaited value in `Right`. Returns `Left(leftValue)` only when the awaited value is `null`.

```
public static Task<Either<TL, TR>> MakeEitherAsync<TL, TR>(this Task<TR> @this,
TL leftValue)
```

Parameters

`this` Task<TR>

A task whose result is wrapped in an LanguageExt.Either<L, R>.

`leftValue` TL

The value used for `Left` when the awaited value is `null`.

Returns

Task<Either<TL, TR>>

A task that resolves to `Right(value)` or `Left(leftValue)`.

Type Parameters

`TL`

The left (error) type.

`TR`

The right (success) type.

MakeEitherAsync<TL, TRInput, TROutput>(Task<TRInput>, Func<TRInput, TROutput>, Predicate<TRInput>, Func<TL>)

Asynchronously applies `map` to the awaited value and wraps the result in `Right` when `leftWhen` returns `false`; otherwise returns `Left` from `leftFunc`.

```
public static Task<Either<TL, TROutput>> MakeEitherAsync<TL, TRInput, TROutput>(this
Task<TRInput> @this, Func<TRInput, TROutput> map, Predicate<TRInput> leftWhen,
Func<TL> leftFunc)
```

Parameters

`this` Task<TRInput>

A task whose result is evaluated and potentially mapped.

`map` Func<TRInput, TROutput>

A projection applied to the awaited value when the result is `Right`.

`leftWhen` Predicate<TRInput>

A predicate evaluated on the awaited value; when `true` the result is `Left`.

`leftFunc` Func<TL>

A factory invoked lazily to produce the `Left` value.

Returns

Task<Either<TL, TROutput>>

A task that resolves to `Right(map(value))` or `Left` from `leftFunc`.

Type Parameters

`TL`

The left (error) type.

`TRInput`

The type of the value produced by the task.

`TROutput`

The type of the mapped right value.

MakeEitherAsync<TL, TRInput, TROutput>(Task<TRInput>, Func<TRInput, TROutput>, Predicate<TRInput>, TL)

Asynchronously applies `map` to the awaited value and wraps the result in `Right` when `leftWhen` returns `false`; otherwise returns `Left(leftValue)`.

```
public static Task<Either<TL, TROutput>> MakeEitherAsync<TL, TRInput, TROutput>
(this Task<TRInput> @this, Func<TRInput, TROutput> map, Predicate<TRInput> leftWhen,
TL leftValue)
```

Parameters

`this` Task<TRInput>

A task whose result is evaluated and potentially mapped.

`map` Func<TRInput, TROutput>

A projection applied to the awaited value when the result is `Right`.

`leftWhen` Predicate<TRInput>

A predicate evaluated on the awaited value; when `true` the result is `Left`.

`leftValue` TL

The value used for `Left` when the predicate is `true`.

Returns

Task<Either<TL, TROutput>>

A task that resolves to `Right(map(value))` or `Left(leftValue)`.

Type Parameters

`TL`

The left (error) type.

`TRInput`

The type of the value produced by the task.

`TROutput`

The type of the mapped right value.

MakeEither<TL, TR>(TR, Func<TL>)

Wraps `this` in `Right`. Returns `Left` produced by invoking `leftFunc` only when the value is `null`.

```
public static Either<TL, TR> MakeEither<TL, TR>(this TR @this, Func<TL> leftFunc)
```

Parameters

`this` TR

The value to wrap.

`leftFunc` Func<TL>

A factory invoked to produce the `Left` value when `this` is `null`.

Returns

Either<TL, TR>

`Right(this)`, or `Left` from `leftFunc` when `this` is `null`.

Type Parameters

`TL`

The left (error) type.

`TR`

The right (success) type.

MakeEither<TL, TR>(TR, Predicate<TR>, Func<TL>)

Wraps `this` in `Right` when `leftWhen` returns `false`; otherwise returns `Left` produced by invoking `leftFunc`.

```
public static Either<TL, TR> MakeEither<TL, TR>(this TR @this, Predicate<TR>
leftWhen, Func<TL> leftFunc)
```

Parameters

`this` TR

The value to evaluate.

`leftWhen` Predicate<TR>

A predicate that, when satisfied, triggers the `Left` case.

`leftFunc` Func<TL>

A factory invoked lazily to produce the `Left` value.

Returns

Either<TL, TR>

`Right(this)`, or `Left` from `leftFunc` when the predicate holds.

Type Parameters

`TL`

The left (error) type.

`TR`

The right (success) type.

MakeEither<TL, TR>(TR, Predicate<TR>, TL)

Wraps `this` in `Right` when `leftWhen` returns `false`; otherwise returns `Left(leftValue)`.

```
public static Either<TL, TR> MakeEither<TL, TR>(this TR @this, Predicate<TR>
leftWhen, TL leftValue)
```

Parameters

`this` TR

The value to evaluate.

`leftWhen` Predicate<TR>

A predicate that, when satisfied, produces the `Left` case.

`leftValue` TL

The value used for `Left` when `leftWhen` is `true`.

Returns

Either<TL, TR>

`Right(this)`, or `Left(leftValue)` when the predicate holds.

Type Parameters

`TL`

The left (error) type.

`TR`

The right (success) type.

MakeEither<TL, TR>(TR, TL)

Wraps `this` in `Right`. Returns `Left(leftValue)` only when the value is `null`.

```
public static Either<TL, TR> MakeEither<TL, TR>(this TR @this, TL leftValue)
```

Parameters

`this` TR

The value to wrap.

`leftValue` TL

The value used for `Left` when `this` is `null`.

Returns

Either<TL, TR>

`Right(this)`, or `Left(leftValue)` when `this` is `null`.

Type Parameters

`TL`

The left (error) type.

`TR`

The right (success) type.

MakeEither<T, TR, TL>(T, Func<T, TR>, Predicate<T>, Func<T, TL>)

Applies `map` to `this` and wraps the result in `Right` when `leftWhen` returns `false`; otherwise returns `Left` produced by applying `leftMap` to the original value.

```
public static Either<TL, TR> MakeEither<T, TR, TL>(this T @this, Func<T, TR> map,
Predicate<T> leftWhen, Func<T, TL> leftMap)
```

Parameters

`this` T

The source value to evaluate and potentially map.

`map` Func<T, TR>

A projection applied to `this` when the result is `Right`.

`leftWhen` Predicate<T>

A predicate evaluated on `this`; when `true` the result is `Left`.

`leftMap` Func<T, TL>

A function applied to `this` to produce the `Left` value.

Returns

Either<TL, TR>

`Right(map(this))`, or `Left(leftMap(this))` when the predicate holds.

Type Parameters

`T`

The type of the source value.

`TR`

The type of the mapped right value.

`TL`

The left (error) type.

MakeEither<T, TR, TL>(T, Func<T, TR>, Predicate<T>, Func<TL>)

Applies `map` to `this` and wraps the result in `Right` when `leftWhen` returns `false`; otherwise returns `Left` from `leftFunc`.

```
public static Either<TL, TR> MakeEither<T, TR, TL>(this T @this, Func<T, TR> map,
Predicate<T> leftWhen, Func<TL> leftFunc)
```

Parameters

`this` T

The source value to evaluate and potentially map.

`map` Func<T, TR>

A projection applied to `this` when the result is `Right`.

`leftWhen` Predicate<T>

A predicate evaluated on `this`; when `true` the result is `Left`.

`leftFunc` Func<TL>

A factory invoked lazily to produce the `Left` value.

Returns

Either<TL, TR>

`Right(map(this))`, or `Left` from `leftFunc` when the predicate holds.

Type Parameters

`T`

The type of the source value.

`TR`

The type of the mapped right value.

`TL`

The left (error) type.

MakeEither<TRInput, TROutput, TL>(TRInput, Func<TRInput, TROutput>, Predicate<TRInput>, TL)

Applies `map` to `this` and wraps the result in `Right` when `leftWhen` returns `false`; otherwise returns `Left(leftValue)`.

```
public static Either<TL, TROutput> MakeEither<TRInput, TROutput, TL>(this TRInput
@this, Func<TRInput, TROutput> map, Predicate<TRInput> leftWhen, TL leftValue)
```

Parameters

`this` TRInput

The source value to evaluate and potentially map.

`map` Func<TRInput, TROutput>

A projection applied to `this` when the result is `Right`.

`leftWhen` Predicate<TRInput>

A predicate evaluated on `this`; when `true` the result is `Left`.

`leftValue` TL

The value used for `Left` when `leftWhen` is `true`.

Returns

Either<TL, TROutput>

`Right(map(this))`, or `Left(leftValue)` when the predicate holds.

Type Parameters

`TRInput`

The type of the source value.

`TROutput`

The type of the mapped right value.

`TL`

The left (error) type.

MakeOptionAsync<T>(Task<T>)

Asynchronously wraps the awaited value in `Some`, returning `None` only when the value is `null`.

```
public static Task<Option<T>> MakeOptionAsync<T>(this Task<T> @this)
```

Parameters

`this` Task<T>

A task whose result is wrapped in an option.

Returns

Task<Option<T>>

A task that resolves to `Some(value)`, or `None` when the awaited value is `null`.

Type Parameters

`T`

The type of the value produced by the task.

MakeOptionAsync<T>(Task<T>, Predicate<T>)

Asynchronously wraps the awaited value in `Some` unless the value is `null` or `noneWhen` returns `true`.

```
public static Task<Option<T>> MakeOptionAsync<T>(this Task<T> @this,
Predicate<T> noneWhen)
```

Parameters

`this` Task<T>

A task whose result is wrapped in an option.

`noneWhen` Predicate<T>

A predicate that, when satisfied, produces `None`.

Returns

Task<Option<T>>

A task that resolves to `Some(value)` or `None` depending on nullability and the predicate.

Type Parameters

`T`

The type of the value produced by the task.

MakeOptionAsync<TInput, TResult>(Task<TInput>, Func<TInput, TResult>, Predicate<TInput>)

Asynchronously applies `map` to the awaited value and wraps the result in `Some`, unless the original value is `null` or `noneWhen` returns `true`.

```
public static Task<Option<TResult>> MakeOptionAsync<TInput, TResult>(this
Task<TInput> @this, Func<TInput, TResult> map, Predicate<TInput> noneWhen)
```

Parameters

`this` Task<TInput>

A task whose result is evaluated and potentially mapped.

`map` Func<TInput, TResult>

A projection applied to the awaited value when the option is `Some`.

`noneWhen` Predicate<TInput>

A predicate evaluated on the awaited value; when `true` the result is `None`.

Returns

Task<Option<TResult>>

A task that resolves to `Some(map(value))`, or `None` when the value is `null` or the predicate holds.

Type Parameters

`TInput`

The type of the value produced by the task.

`TResult`

The type of the mapped value stored inside the option.

MakeOption<T>(T)

Wraps `this` in `Some`, returning `None` only when the value is `null`.

```
public static Option<T> MakeOption<T>(this T @this)
```

Parameters

`this` T

The value to wrap.

Returns

Option<T>

`Some(this)`, or `None` when `this` is `null`.

Type Parameters

`T`

The type of the value.

MakeOption<T>(T, Predicate<T>)

Wraps `this` in `Some` unless the value is `null` or `noneWhen` returns `true`.

```
public static Option<T> MakeOption<T>(this T @this, Predicate<T> noneWhen)
```

Parameters

`this` T

The value to wrap.

`noneWhen` Predicate<T>

A predicate that, when satisfied, produces `None` instead of `Some`.

Returns

Option<T>

`Some(this)`, or `None` when the value is `null` or `noneWhen` returns `true`.

Type Parameters

`T`

The type of the value.

MakeOption<TInput, TResult>(TInput, Func<TInput, TResult>, Predicate<TInput>)

Applies `map` to `this` and wraps the result in `Some`, unless the original value is `null` (for reference types) or `noneWhen` returns `true`.

```
public static Option<TResult> MakeOption<TInput, TResult>(this TInput @this,
Func<TInput, TResult> map, Predicate<TInput> noneWhen)
```

Parameters

`this` TInput

The source value to evaluate and potentially map.

`map` Func<TInput, TResult>

A projection applied to `this` when the option is `Some`.

`noneWhen` Predicate<TInput>

A predicate evaluated on `this`; when `true` the result is `None`.

Returns

Option<TResult>

`Some(map(this))`, or `None` when the value is `null` or the predicate holds.

Type Parameters

`TInput`

The type of the source value.

`TResult`

The type of the mapped value stored inside the option.

MapAsync<TSource, TResult>(Task<TSource>, Func<TSource, Task<TResult>>)

Awaits `this`, then passes the result to the asynchronous function `fn` and awaits the produced task.

```
public static Task<TResult> MapAsync<TSource, TResult>(this Task<TSource> @this,
Func<TSource, Task<TResult>> fn)
```

Parameters

`this` Task<TSource>

A task whose result is passed to `fn`.

`fn` Func<TSource, Task<TResult>>

An asynchronous transformation function.

Returns

Task<TResult>

A task that resolves to the result of `fn`.

Type Parameters

`TSource`

The type of the value produced by `this`.

`TResult`

The type of the value produced by `fn`.

MapAsync<TSource, TResult>(Task<TSource>, Func<TSource, TResult>)

Awaits `this` and applies the synchronous function `fn` to the result.

```
public static Task<TResult> MapAsync<TSource, TResult>(this Task<TSource> @this,
Func<TSource, TResult> fn)
```

Parameters

`this` Task<TSource>

A task whose result is passed to `fn`.

`fn` Func<TSource, TResult>

A synchronous transformation function applied to the awaited value.

Returns

Task<TResult>

A task that resolves to the result of `fn`.

Type Parameters

`TSource`

The type of the value produced by `this`.

`TResult`

The type of the value produced by `fn`.

MapAsync<TSource, TResult>(TSource, Func<TSource, Task<TResult>>)

Passes `this` directly to the asynchronous function `fn`.

```
public static Task<TResult> MapAsync<TSource, TResult>(this TSource @this,
Func<TSource, Task<TResult>> fn)
```

Parameters

`this` TSource

The value to pass to `fn`.

`fn` Func<TSource, Task<TResult>>

An asynchronous transformation function.

Returns

Task<TResult>

A task that resolves to the result of `fn`.

Type Parameters

`TSource`

The type of the source value.

`TResult`

The type of the value produced by `fn`.

MapLeftAsync<TL, TR, TMl>(Task<Either<TL, TR>>, Func<TL, Task<TMl>>)

Asynchronously maps the `Left` side of an LanguageExt.Either<L, R> using an async function, leaving any `Right` value unchanged.

```
public static Task<Either<TMl, TR>> MapLeftAsync<TL, TR, TMl>(this Task<Either<TL,
TR>> @this, Func<TL, Task<TMl>> onLeftAsync)
```

Parameters

`this` Task<Either<TL, TR>>

A task that resolves to an LanguageExt.Either<L, R>.

`onLeftAsync` Func<TL, Task<TMl>>

An async function that transforms the left value.

Returns

Task<Either<TMl, TR>>

A task that resolves to an LanguageExt.Either<L, R> with the left type remapped.

Type Parameters

`TL`

The original left type.

`TR`

The right type, unchanged by this operation.

`TMl`

The new left type produced by `onLeftAsync`.

MapLeftAsync<TL, TR, TMl>(Task<Either<TL, TR>>, Func<TL, TMl>)

Asynchronously maps the `Left` side of an LanguageExt.Either<L, R> using a synchronous function, leaving any `Right` value unchanged.

```
public static Task<Either<TMl, TR>> MapLeftAsync<TL, TR, TMl>(this Task<Either<TL,
TR>> @this, Func<TL, TMl> onLeft)
```

Parameters

`this` Task<Either<TL, TR>>

A task that resolves to an LanguageExt.Either<L, R>.

`onLeft` Func<TL, TMl>

A synchronous function that transforms the left value.

Returns

Task<Either<TMl, TR>>

A task that resolves to an LanguageExt.Either<L, R> with the left type remapped.

Type Parameters

`TL`

The original left type.

`TR`

The right type, unchanged by this operation.

`TMl`

The new left type produced by `onLeft`.

Map<TSource, TResult>(TSource, Func<TSource, TResult>)

Applies `fn` to `this`, enabling fluent transformation pipelines.

```
public static TResult Map<TSource, TResult>(this TSource @this, Func<TSource,
TResult> fn)
```

Parameters

`this` TSource

The value to transform.

`fn` Func<TSource, TResult>

The transformation function to apply.

Returns

TResult

The result of applying `fn` to `this`.

Type Parameters

`TSource`

The type of the source value.

`TResult`

The type of the result produced by `fn`.

MatchAsync<TL, TR, TOutput>(Task<Either<TL, TR>>, Func<TR, Task<TOutput>>, Func<TL, Task<TOutput>>)

Asynchronously pattern-matches an LanguageExt.Either<L, R> by awaiting both branch functions.

```
public static Task<TOutput> MatchAsync<TL, TR, TOutput>(this Task<Either<TL, TR>>
@this, Func<TR, Task<TOutput>> onRightAsync, Func<TL, Task<TOutput>> onLeftAsync)
```

Parameters

`this` Task<Either<TL, TR>>

A task that resolves to an LanguageExt.Either<L, R>.

`onRightAsync` Func<TR, Task<TOutput>>

An async function invoked when the either is `Right`.

`onLeftAsync` Func<TL, Task<TOutput>>

An async function invoked when the either is `Left`.

Returns

Task<TOutput>

A task that resolves to the output of whichever branch was taken.

Type Parameters

`TL`

The left (error) type.

TR

The right (success) type.

TOutput

The type produced by both branch functions.

MatchAsync<TL, TR, TOutput>(Task<Either<TL, TR>>, Func<TR, Task<TOutput>>, Func<TL, TOutput>)

Asynchronously pattern-matches an LanguageExt.Either<L, R> where only the right branch is async.

```
public static Task<TOutput> MatchAsync<TL, TR, TOutput>(this Task<Either<TL, TR>>
@this, Func<TR, Task<TOutput>> onRightAsync, Func<TL, TOutput> onLeft)
```

Parameters

`this` Task<Either<TL, TR>>

A task that resolves to an LanguageExt.Either<L, R>.

`onRightAsync` Func<TR, Task<TOutput>>

An async function invoked when the either is `Right`.

`onLeft` Func<TL, TOutput>

A synchronous function invoked when the either is `Left`.

Returns

Task<TOutput>

A task that resolves to the output of whichever branch was taken.

Type Parameters

TL

The left (error) type.

TR

The right (success) type.

TOutput

The type produced by both branch functions.

MatchAsync<TL, TR, TOutput>(Task<Either<TL, TR>>, Func<TR, TOutput>, Func<TL, Task<TOutput>>)

Asynchronously pattern-matches an LanguageExt.Either<L, R> where only the left branch is async.

```
public static Task<TOutput> MatchAsync<TL, TR, TOutput>(this Task<Either<TL, TR>>
@this, Func<TR, TOutput> onRight, Func<TL, Task<TOutput>> onLeftAsync)
```

Parameters

this Task<Either<TL, TR>>

A task that resolves to an LanguageExt.Either<L, R>.

onRight Func<TR, TOutput>

A synchronous function invoked when the either is `Right`.

onLeftAsync Func<TL, Task<TOutput>>

An async function invoked when the either is `Left`.

Returns

Task<TOutput>

A task that resolves to the output of whichever branch was taken.

Type Parameters

TL

The left (error) type.

`TR`

The right (success) type.

`TOutput`

The type produced by both branch functions.

MatchAsync<TL, TR, TOutput>(Task<Either<TL, TR>>, Func<TR, TOutput>, Func<TL, TOutput>)

Awaits the LanguageExt.Either<L, R> and synchronously pattern-matches it, invoking the appropriate branch function and returning the result.

```
public static Task<TOutput> MatchAsync<TL, TR, TOutput>(this Task<Either<TL, TR>>
@this, Func<TR, TOutput> onRight, Func<TL, TOutput> onLeft)
```

Parameters

`this` Task<Either<TL, TR>>

A task that resolves to an LanguageExt.Either<L, R>.

`onRight` Func<TR, TOutput>

A synchronous function invoked when the either is `Right`.

`onLeft` Func<TL, TOutput>

A synchronous function invoked when the either is `Left`.

Returns

Task<TOutput>

A task that resolves to the output of whichever branch was taken.

Type Parameters

`TL`

The left (error) type.

`TR`

The right (success) type.

`TOutput`

The type produced by both branch functions.

MatchUnsafeAsync<TL, TR, TOutput>(Task<Either<TL, TR>>, Func<TR, TOutput>, Func<TL, TOutput>)

Awaits the LanguageExt.Either<L, R> and synchronously pattern-matches it using `MatchUnsafe`, which allows `null` return values from branch functions.

```
public static Task<TOutput> MatchUnsafeAsync<TL, TR, TOutput>(this Task<Either<TL,
TR>> @this, Func<TR, TOutput> onRight, Func<TL, TOutput> onLeft)
```

Parameters

`this` Task<Either<TL, TR>>

A task that resolves to an LanguageExt.Either<L, R>.

`onRight` Func<TR, TOutput>

A function invoked when the either is `Right`; may return `null`.

`onLeft` Func<TL, TOutput>

A function invoked when the either is `Left`; may return `null`.

Returns

Task<TOutput>

A task that resolves to the output of whichever branch was taken, potentially `null`.

Type Parameters

`TL`

The left (error) type.

`TR`

The right (success) type.

`TOutput`

The type produced by both branch functions. May be `null`.

OrElse<T>(Option<T>, T)

Extracts the value from `this` when it is `Some`, or returns `defaultValue` when it is `None`.

```
public static T OrElse<T>(this Option<T> @this, T defaultValue)
```

Parameters

`this` Option<T>

The option to extract a value from.

`defaultValue` T

The fallback value returned when `this` is `None`.

Returns

T

The inner value if `Some`; otherwise `defaultValue`.

Type Parameters

`T`

The type of the optional value.

SameMap<TSource, TResult>((TSource, TSource), Func<TSource, TResult>)

Applies the same function `fn` to both elements of a same-typed tuple.

```
public static (TResult, TResult) SameMap<TSource, TResult>(this (TSource, TSource)
@this, Func<TSource, TResult> fn)
```

Parameters

`this` (TSource, TSource)

A tuple whose two elements are both of type `TSource`.

`fn` Func<TSource, TResult>

The function applied independently to each element.

Returns

(TResult, TResult)

A new tuple containing `(fn(Item1), fn(Item2))`.

Type Parameters

`TSource`

The element type of the source tuple.

`TResult`

The element type of the result tuple.

TeeWhenAsync<T>(Task<T>, Func<T, Task>, Func<T, bool>)

Awaits the task, then asynchronously invokes `tee` on the result when `when` returns `true`, returning the original value.

```
public static Task<T> TeeWhenAsync<T>(this Task<T> thistask, Func<T, Task> tee,
Func<T, bool> when)
```

Parameters

thistask Task<T>

A task whose result is evaluated.

tee Func<T, Task>

An async side-effecting function invoked conditionally.

when Func<T, bool>

A predicate evaluated on the awaited value to decide whether to invoke `tee`.

Returns

Task<T>

A task that resolves to the awaited value unchanged.

Type Parameters

T

The type of the value produced by the task.

TeeWhenAsync<T>(T, Func<T, Task<T>>, Func<T, bool>)

Invokes the async transformation `tee` when `when` returns `true`, returning a task that resolves to the transformed value; otherwise resolves to the original value.

```
public static Task<T> TeeWhenAsync<T>(this T @this, Func<T, Task<T>> tee, Func<T,
bool> when)
```

Parameters

this T

The value to pass through.

tee Func<T, Task<T>>

An async function that transforms the value.

`when` Func<T, bool>

A predicate evaluated on `this` to decide whether to call `tee`.

Returns

Task<T>

A task that resolves to the result of `tee` when the predicate holds; otherwise `this` wrapped in a completed task.

Type Parameters

`T`

The type of the value.

TeeWhenAsync<T>(T, Func<T, Task>, bool)

Asynchronously invokes `tee` on `this` when `when` is `true`, then returns the original value.

```
public static Task<T> TeeWhenAsync<T>(this T @this, Func<T, Task> tee, bool when)
```

Parameters

`this` T

The value to pass through.

`tee` Func<T, Task>

An async side-effecting function invoked conditionally.

`when` bool

A boolean flag that controls whether `tee` is awaited.

Returns

Task<T>

A task that resolves to `this` unchanged.

Type Parameters

`T`

The type of the value.

TeeWhenAsync<T>(T, Func<T, Task>, Func<T, bool>)

Asynchronously invokes `tee` on `this` when `when` returns `true` for the current value, then returns the original value.

```
public static Task<T> TeeWhenAsync<T>(this T @this, Func<T, Task> tee, Func<T,
bool> when)
```

Parameters

`this` T

The value to pass through.

`tee` Func<T, Task>

An async side-effecting function invoked conditionally.

`when` Func<T, bool>

A predicate evaluated on `this` to decide whether to invoke `tee`.

Returns

Task<T>

A task that resolves to `this` unchanged.

Type Parameters

`T`

The type of the value.

TeeWhen<T>(T, Action<T>, Func<bool>)

Invokes the side-effecting `tee` action on `this` when `when` returns `true`, then returns the original value unchanged.

```
public static T TeeWhen<T>(this T @this, Action<T> tee, Func<bool> when)
```

Parameters

`this` T

The value to pass through.

`tee` Action<T>

A side-effecting action invoked conditionally on the value.

`when` Func<bool>

A parameterless predicate that controls whether `tee` is invoked.

Returns

T

`this` unchanged, regardless of whether the action ran.

Type Parameters

`T`

The type of the value.

TeeWhen<T>(T, Func<T, T>, bool)

Applies `tee` to `this` when `when` is `true`, returning the original value either way.

```
public static T TeeWhen<T>(this T @this, Func<T, T> tee, bool when)
```

Parameters

`this` T

The value to pass through.

`tee` Func<T, T>

A function that receives the value and returns a transformed copy.

`when` bool

A boolean flag that controls whether `tee` is invoked.

Returns

T

The result of `tee` when `when` is `true`; otherwise `this` unchanged.

Type Parameters

`T`

The type of the value.

TeeWhen<T>(T, Func<T, T>, Func<bool>)

Applies `tee` to `this` when the parameterless predicate `when` returns `true`, returning the original value either way.

```
public static T TeeWhen<T>(this T @this, Func<T, T> tee, Func<bool> when)
```

Parameters

`this` T

The value to pass through.

`tee` Func<T, T>

A function that receives the value and returns a transformed copy.

`when` Func<bool>

A parameterless predicate that controls whether `tee` is invoked.

Returns

T

The result of `tee` when the predicate holds; otherwise `this` unchanged.

Type Parameters

`T`

The type of the value.

TeeWhen<T>(T, Func<T, T>, Func<T, bool>)

Applies `tee` to `this` when `when` returns `true` for the current value, returning the original value either way.

```
public static T TeeWhen<T>(this T @this, Func<T, T> tee, Func<T, bool> when)
```

Parameters

`this` T

The value to pass through.

`tee` Func<T, T>

A function that receives the value and returns a transformed copy.

`when` Func<T, bool>

A predicate evaluated on `this` to decide whether to apply `tee`.

Returns

T

The result of `tee` when the predicate holds; otherwise `this` unchanged.

Type Parameters

T

The type of the value.

Tee<T>(T, Action)

Invokes the parameterless side-effecting `tee` action and returns `this` unchanged.

```
public static T Tee<T>(this T @this, Action tee)
```

Parameters

`this` T

The value to pass through.

`tee` Action

A parameterless side-effecting action.

Returns

T

`this` unchanged after the action has run.

Type Parameters

T

The type of the value.

Tee<T>(T, Action<T>)

Invokes the side-effecting `tee` action on `this` and returns `this` unchanged, enabling pass-through pipelines.

```
public static T Tee<T>(this T @this, Action<T> tee)
```

Parameters

`this` T

The value to pass through.

`tee` Action<T>

A side-effecting action invoked on the value.

Returns

T

`this` unchanged after the action has run.

Type Parameters

`T`

The type of the value.

Tee<T>(T, Func<T, T>)

Applies `tee` to `this` unconditionally and returns the result.

```
public static T Tee<T>(this T @this, Func<T, T> tee)
```

Parameters

`this` T

The value to transform.

`tee` Func<T, T>

A function that receives the value and returns a transformed copy.

Returns

T

The result of applying `tee` to `this`.

Type Parameters

`T`

The type of the value.

UsingAsync<TD>(TD, Func<TD, Task>)

Asynchronously executes `action` with `disposable`, disposes it, and returns LanguageExt.Unit.

```
public static Task<Unit> UsingAsync<TD>(TD disposable, Func<TD, Task> action) where
TD : IDisposable
```

Parameters

`disposable` TD

The resource to use and then dispose.

`action` Func<TD, Task>

An async side-effecting function to execute with `disposable`.

Returns

Task<Unit>

A task that resolves to LanguageExt.Unit.Default after the action and disposal complete.

Type Parameters

`TD`

The disposable type.

UsingAsync<TD, T>(TD, Func<TD, Task<T>>)

Asynchronously executes `func` with `disposable`, disposes it, and returns the result.

```
public static Task<T> UsingAsync<TD, T>(TD disposable, Func<TD, Task<T>> func) where
TD : IDisposable
```

Parameters

`disposable` TD

The resource to use and then dispose.

`func` Func<TD, Task<T>>

An async function that produces a value from `disposable`.

Returns

Task<T>

A task that resolves to the value returned by `func`.

Type Parameters

`TD`

The disposable type.

`T`

The type of the value produced by `func`.

UsingAsync<TD1, TD2>(TD1, Func<TD1, TD2>, Func<TD1, TD2, Task>)

Asynchronously creates a second disposable from the first, executes `action` with both, disposes them in reverse order, and returns LanguageExt.Unit.

```
public static Task<Unit> UsingAsync<TD1, TD2>(TD1 disposable1, Func<TD1, TD2>
createDisposable2, Func<TD1, TD2, Task> action) where TD1 : IDisposable where TD2
: IDisposable
```

Parameters

disposable1 TD1

The primary resource.

createDisposable2 Func<TD1, TD2>

A factory that derives a second disposable from disposable1.

action Func<TD1, TD2, Task>

An async side-effecting function executed with both resources.

Returns

Task<Unit>

A task that resolves to LanguageExt.Unit.Default after both resources are disposed.

Type Parameters

TD1

The type of the first disposable.

TD2

The type of the second disposable, derived from disposable1.

UsingAsync<TD1, TD2, T>(TD1, Func<TD1, TD2>, Func<TD1, TD2, Task<T>>)

Asynchronously creates a second disposable from the first, executes func with both, disposes them in reverse order, and returns the result.

```
public static Task<T> UsingAsync<TD1, TD2, T>(TD1 disposable1, Func<TD1, TD2>
createDisposable2, Func<TD1, TD2, Task<T>> func) where TD1 : IDisposable where TD2
: IDisposable
```

Parameters

disposable1 TD1

The primary resource.

`createDisposable2` Func<TD1, TD2>

A factory that derives a second disposable from `disposable1`.

`func` Func<TD1, TD2, Task<T>>

An async function that produces a value from both disposables.

Returns

Task<T>

A task that resolves to the value returned by `func`.

Type Parameters

`TD1`

The type of the first disposable.

`TD2`

The type of the second disposable, derived from `disposable1`.

`T`

The type of the value produced by `func`.

Using<TD>(TD, Action<TD>)

Executes `action` with `disposable`, disposing it afterwards, and returns LanguageExt.Unit.

```
public static Unit Using<TD>(TD disposable, Action<TD> action) where TD
: IDisposable
```

Parameters

`disposable` TD

The resource to use and then dispose.

`action` Action<TD>

The side-effecting action to execute with `disposable`.

Returns

Unit

LanguageExt.Unit.Default after the action completes and the resource is disposed.

Type Parameters

`TD`

The disposable type.

Using<TD, T>(TD, Func<TD, T>)

Executes `func` with `disposable`, disposes it, and returns the result.

```
public static T Using<TD, T>(TD disposable, Func<TD, T> func) where TD : IDisposable
```

Parameters

`disposable` TD

The resource to use and then dispose.

`func` Func<TD, T>

A function that produces a value from `disposable`.

Returns

T

The value returned by `func`.

Type Parameters

`TD`

The disposable type.

`T`

The type of the value returned by `func`.

Using<TD1, TD2>(TD1, Func<TD1, TD2>, Action<TD1, TD2>)

Creates a second disposable from the first using `createDisposable2`, executes `action` with both, disposes them in reverse order, and returns LanguageExt.Unit.

```
public static Unit Using<TD1, TD2>(TD1 disposable1, Func<TD1, TD2>
createDisposable2, Action<TD1, TD2> action) where TD1 : IDisposable where TD2
: IDisposable
```

Parameters

`disposable1` TD1

The primary resource.

`createDisposable2` Func<TD1, TD2>

A factory that derives a second disposable from `disposable1`.

`action` Action<TD1, TD2>

The side-effecting action to execute with both resources.

Returns

Unit

LanguageExt.Unit.Default after both resources have been disposed.

Type Parameters

`TD1`

The type of the first disposable.

TD2

The type of the second disposable, created from disposable1.

Using<TD1, TD2, T>(TD1, Func<TD1, TD2>, Func<TD1, TD2, T>)

Creates a second disposable from the first, executes func with both, disposes them in reverse order, and returns the result.

```
public static T Using<TD1, TD2, T>(TD1 disposable1, Func<TD1, TD2>
createDisposable2, Func<TD1, TD2, T> func) where TD1 : IDisposable where TD2
: IDisposable
```

Parameters

disposable1 TD1

The primary resource.

createDisposable2 Func<TD1, TD2>

A factory that derives a second disposable from disposable1.

func Func<TD1, TD2, T>

A function that produces a value from both disposables.

Returns

T

The value returned by func.

Type Parameters

TD1

The type of the first disposable.

TD2

The type of the second disposable, created from `disposable1`.

`T`

The type of the value returned by `func`.

Namespace Fl.Functional.Utils.Recursion

Classes

RecursionResult<T>

TailRecursion

Class RecursionResult<T>

Namespace: FI.Functional.Utils.Recursion

Assembly: FI.Functional.Utils.dll

```
public class RecursionResult<T>
```

Type Parameters

T

Inheritance

object ← RecursionResult<T>

Inherited Members

object.Equals(object) , object.Equals(object, object) , object.GetHashCode() , object.GetType() , object.MemberwiseClone() , object.ReferenceEquals(object, object) , object.ToString()

Extension Methods

Functional.DoAsync<T>(T, Func<T, Task>) , Functional.Do<T>(T, Action<T>) , Functional.MakeEither<TL, TR>(TR, Func<TL>) , Functional.MakeEither<TL, TR>(TR, Predicate<TR>, Func<TL>) , Functional.MakeEither<TL, TR>(TR, Predicate<TR>, TL) , Functional.MakeEither<TL, TR>(TR, TL) , Functional.MakeEither<TRInput, TROutput, TL>(TRInput, Func<TRInput, TROutput>, Predicate<TRInput>, TL) , Functional.MakeEither<T, TR, TL>(T, Func<T, TR>, Predicate<T>, Func<T, TL>) , Functional.MakeEither<T, TR, TL>(T, Func<T, TR>, Predicate<T>, Func<TL>) , Functional.MakeOption<T>(T) , Functional.MakeOption<T>(T, Predicate<T>) , Functional.MakeOption<TInput, TResult>(TInput, Func<TInput, TResult>, Predicate<TInput>) , Functional.MapAsync<TSource, TResult>(TSource, Func<TSource, Task<TResult>>) , Functional.Map<TSource, TResult>(TSource, Func<TSource, TResult>) , Functional.TeeWhenAsync<T>(T, Func<T, Task<T>>, Func<T, bool>) , Functional.TeeWhenAsync<T>(T, Func<T, Task>, bool) , Functional.TeeWhenAsync<T>(T, Func<T, Task>, Func<T, bool>) , Functional.TeeWhen<T>(T, Action<T>, Func<bool>) , Functional.TeeWhen<T>(T, Func<T, T>, bool) , Functional.TeeWhen<T>(T, Func<T, T>, Func<T, bool>) , Functional.TeeWhen<T>(T, Func<T, T>, Func<bool>) , Functional.Tee<T>(T, Action) , Functional.Tee<T>(T, Action<T>) , Functional.Tee<T>(T, Func<T, T>)

Properties

IsFinalResult

Gets a value indicating whether this is the terminal result of the recursive computation. When `true`, the trampoline stops and returns Result.

```
public bool IsFinalResult { get; }
```

Property Value

bool

NextStep

Gets the factory for the next synchronous recursion step. Used by the trampoline when IsFinalResult is `false`.

```
public Func<RecursionResult<T>> NextStep { get; }
```

Property Value

Func<RecursionResult<T>>

NextStepAsync

Gets the factory for the next async recursion step. Used by the trampoline when IsFinalResult is `false`.

```
public Func<Task<RecursionResult<T>>> NextStepAsync { get; }
```

Property Value

Func<Task<RecursionResult<T>>>

Result

Gets the value carried by this recursion step. Only meaningful as the final output when IsFinalResult is `true`.

```
public T Result { get; }
```

Property Value

T

Methods

CreateLast(T, Func<RecursionResult<T>>)

Creates a terminal synchronous recursion result that signals the trampoline to stop and return `result` as the final value.

```
public static RecursionResult<T> CreateLast(T result, Func<RecursionResult<T>>
nextStep)
```

Parameters

`result` T

The final value of the recursive computation.

`nextStep` Func<RecursionResult<T>>

Ignored by the trampoline; may be `null`.

Returns

RecursionResult<T>

A RecursionResult<T> with IsFinalResult set to `true`.

CreateLastAsync(T, Func<Task<RecursionResult<T>>>)

Creates a terminal async recursion result that signals the trampoline to stop and return `result` as the final value.

```
public static Task<RecursionResult<T>> CreateLastAsync(T result,
Func<Task<RecursionResult<T>>> nextStep)
```

Parameters

`result` T

The final value of the recursive computation.

`nextStep` Func<Task<RecursionResult<T>>>

Ignored by the trampoline; may be `null`.

Returns

Task<RecursionResult<T>>

A task that resolves to a RecursionResult<T> with IsFinalResult set to `true`.

CreateNext(T, Func<RecursionResult<T>>)

Creates an intermediate synchronous recursion result that signals the trampoline to continue by invoking `nextStep` on the next iteration.

```
public static RecursionResult<T> CreateNext(T result, Func<RecursionResult<T>>
nextStep)
```

Parameters

`result` T

An intermediate value (typically `default`); not used as the final result.

`nextStep` Func<RecursionResult<T>>

A factory that produces the following synchronous recursion step.

Returns

RecursionResult<T>

A RecursionResult<T> with IsFinalResult set to `false`.

CreateNextAsync(T, Func<Task<RecursionResult<T>>>)

Creates an intermediate async recursion result that signals the trampoline to continue by invoking `nextStep` on the next iteration.

```
public static Task<RecursionResult<T>> CreateNextAsync(T result,
Func<Task<RecursionResult<T>>> nextStep)
```

Parameters

`result` T

An intermediate value (typically `default`); not used as the final result.

`nextStep` Func<Task<RecursionResult<T>>>

A factory that produces the following async recursion step.

Returns

Task<RecursionResult<T>>

A task that resolves to a RecursionResult<T> with IsFinalResult set to `false`.

Class TailRecursion

Namespace: FI.Functional.Utils.Recursion

Assembly: FI.Functional.Utils.dll

```
public static class TailRecursion
```

Inheritance

object ← TailRecursion

Inherited Members

object.Equals(object) , object.Equals(object, object) , object.GetHashCode() , object.GetType() , object.MemberwiseClone() , object.ReferenceEquals(object, object) , object.ToString()

Methods

ExecuteAsync<T>(Func<Task<RecursionResult<T>>>)

Executes an asynchronous tail-recursive computation by repeatedly invoking `func` until a RecursionResult<T> with IsFinalResult set to `true` is returned, avoiding stack overflows via a trampoline loop.

```
public static Task<T> ExecuteAsync<T>(Func<Task<RecursionResult<T>>> func)
```

Parameters

`func` Func<Task<RecursionResult<T>>>

A factory that produces the next async recursion step.

Returns

Task<T>

A task that resolves to the final result of the recursion.

Type Parameters

T

The type of the final result.

Execute<T>(Func<RecursionResult<T>>)

Executes a synchronous tail-recursive computation by repeatedly invoking `func` until a RecursionResult<T> with IsFinalResult set to `true` is returned, avoiding stack overflows via a trampoline loop.

```
public static T Execute<T>(Func<RecursionResult<T>> func)
```

Parameters

`func` Func<RecursionResult<T>>

A factory that produces the next recursion step.

Returns

T

The final result of the recursive computation.

Type Parameters

T

The type of the final result.

NextAsync<T>(Func<Task<RecursionResult<T>>>)

Creates an intermediate RecursionResult<T> that continues the async recursion by invoking `nextStep` on the next iteration.

```
public static Task<RecursionResult<T>> NextAsync<T>(Func<Task<RecursionResult<T>>>
nextStep)
```

Parameters

`nextStep` Func<Task<RecursionResult<T>>>

A factory producing the following recursion step.

Returns

Task<RecursionResult<T>>

A task that resolves to a RecursionResult<T> pointing to the next step.

Type Parameters

`T`

The type of the eventual result.

Next<T>(Func<RecursionResult<T>>)

Creates an intermediate RecursionResult<T> that continues the synchronous recursion by invoking `nextStep` on the next iteration.

```
public static RecursionResult<T> Next<T>(Func<RecursionResult<T>> nextStep)
```

Parameters

`nextStep` Func<RecursionResult<T>>

A factory producing the following recursion step.

Returns

RecursionResult<T>

A RecursionResult<T> pointing to the next step.

Type Parameters

`T`

The type of the eventual result.

ReturnAsync<T>(T)

Creates a terminal RecursionResult<T> that signals the end of an async recursion, wrapping `result` as the final value.

```
public static Task<RecursionResult<T>> ReturnAsync<T>(T result)
```

Parameters

`result` T

The final value to return from the recursive computation.

Returns

Task<RecursionResult<T>>

A task that resolves to a RecursionResult<T> marked as the final result.

Type Parameters

`T`

The type of the result.

Return<T>(T)

Creates a terminal RecursionResult<T> that signals the end of a synchronous recursion, wrapping `result` as the final value.

```
public static RecursionResult<T> Return<T>(T result)
```

Parameters

`result` T

The final value to return from the recursive computation.

Returns

RecursionResult<T>

A RecursionResult<T> marked as the final result.

Type Parameters

T

The type of the result.