
Effect

Release 0.10

August 25, 2015

1	Documentation	3
1.1	Quick Introduction	3
1.2	Testing Effectful Code	5
1.3	API documentation	5
2	Indices and tables	17
	Python Module Index	19

Effect is a library for helping you write purely functional code by isolating the effects (that is, IO or state manipulation) in your code.

It supports both Python 2.6 and up, and 3.4 and up, as well as PyPy.

It lives on PyPI at <https://pypi.python.org/pypi/effect> and GitHub at <https://github.com/python-effect/effect>.

Documentation

1.1 Quick Introduction

1.1.1 Explanation by Example

Effect starts with a very simple idea: instead of having a function which performs side-effects (such as IO):

```
def get_user_name():
    return raw_input("Enter User Name> ") # or 'input' in Python 3
```

you instead have a function which *returns* a representation of the side-effect:

```
def get_user_name():
    return Effect(ReadLine("Enter User Name> "))
```

We call objects like `ReadLine` an *intent* – that is, the *intent* of this effect is to read a line of input from the user. Ideally, intents are very simple objects with public attributes and no behavior, only data.

```
class ReadLine(object):
    def __init__(self, prompt):
        self.prompt = prompt
```

To perform the `ReadLine` intent, we must implement a performer function:

```
@sync_performer
def perform_read_line(dispatcher, readline):
    return raw_input(readline.prompt)
```

To do something with the result of the effect, we must attach callbacks with the `on` method:

```
def greet():
    return get_user_name().on(
        success=lambda r: Effect(Print("Hello,", r)),
        error=lambda exc_info: Effect(Print("There was an error!", exc_info[1])))
```

(Here we assume another intent, `Print`, which shows some text to the user.)

A (sometimes) nicer syntax is provided for adding callbacks, with the `effect.do.do()` decorator.

```
from effect.do import do

@do
def greet():
    try:
```

```
name = yield get_user_name()
except Exception as e:
    yield Effect(Print("There was an error!", e))
else:
    yield Effect(Print("Hello, ", name))
```

Finally, to actually perform these effects, they can be passed to `effect.sync_perform()`, along with a dispatcher which looks up the performer based on the intent.

```
from effect import sync_perform

def main():
    eff = greet()
    dispatcher = ComposedDispatcher([
        TypeDispatcher({ReadLine: perform_read_line}),
        base_dispatcher])
    sync_perform(dispatcher, eff)
```

This has a number of advantages. First, your unit tests for `get_user_name` become simpler. You don't need to mock out or parameterize the `raw_input` function - you just call `get_user_name` and assert that it returns a `ReadLine` object with the correct 'prompt' value.

Second, you can implement `ReadLine` in a number of different ways - it's possible to override the way an intent is performed to do whatever you want. For example, you could implement an `HTTPRequest` client either using the popular `requests` package, or using the Twisted-based `treq` package – without needing to change any of your application code, since it's all in terms of the Effect API.

1.1.2 A quick tour, with definitions

- **Intent:** An object which describes a desired action, ideally with simple inert data in public attributes. For example, `ReadLine(prompt='> ')` could be an intent that describes the desire to read a line from the user after showing a prompt.
- `effect.Effect`: An object which binds callbacks to receive the result of performing an intent.
- **Performer:** A callable that takes the Dispatcher, an Intent, and a Box. It executes the Intent and puts the result in the Box. For example, the performer for `ReadLine()` could call `raw_input(intent.prompt)`.
- **Dispatcher:** A callable that takes an Intent and finds the Performer that can execute it (or None). See `TypeDispatcher` and `ComposedDispatcher` for handy pre-built dispatchers.
- **Box:** An object that has `succeed` and `fail` methods for providing the result of an effect (potentially asynchronously). Usually you don't need to care about this, if you define your performers with `effect.sync_performer()` or `txeffect.deferred_performer` from the `txeffect` package.

There's a few main things you need to do to use Effect.

- Define some intents to describe your side-effects (or use a library containing intents that already exist). For example, an `HTTPRequest` intent that has `method`, `url`, etc attributes.
- Write your application code to create effects like `Effect(HTTPRequest(...))` and attach callbacks to them with `Effect.on()`.
- As close as possible to the top-level of your application, perform your effect(s) with `effect.sync_perform()`.
- You will need to pass a dispatcher to `effect.sync_perform()`. You should create one by creating a `effect.TypeDispatcher` with your own performers (e.g. for `HTTPRequest`), and

composing it with `effect.base_dispatcher` (which has performers for built-in effects) using `effect.ComposedDispatcher`.

1.1.3 Callback chains

Effect allows you to build up chains of callbacks that process data in turn. That is, if you attach a callback `a` and then a callback `b` to an Effect, `a` will be called with the original result, and `b` will be called with the result of `a`. This is exactly how Twisted's Deferreds work, and similar to the monadic `bind(>=>)` function from Haskell.

This is a great way to build abstractions, compared to non-chaining callback systems like Python's Futures. You can easily build abstractions like the following:

```
def request_url(method, url, str_body):
    """Perform an HTTP request."""
    return Effect(HTTPRequest(method, url, str_body))

def request_200_url(method, url, str_body):
    """
    Perform an HTTP request, and raise an error if the response is not 200.
    """
    def check_status(response):
        if response.code != 200:
            raise HTTPError(response.code)
        return response
    return request_url(method, url, str_body).on(success=check_status)

def json_request(method, url, dict_body):
    """
    Perform an HTTP request where the body is sent as JSON and the response
    is automatically decoded as JSON if the Content-type is
    application/json.
    """
    str_body = json.dumps(dict_body)
    return request_200_url(method, url, str_body).on(success=decode_json)
```

1.2 Testing Effectful Code

The most useful testing tool you'll want to familiarize yourself with is `effect.testing.perform_sequence()`. Using this in your unit tests will allow you to perform your effects while ensuring that the expected intents are performed in the expected order, as well as provide the results of those effects.

1.3 API documentation

1.3.1 Core API

A system for helping you separate your IO and state-manipulation code (hereafter referred to as “effects”) from everything else, thus allowing the majority of your code to be easier to test and compose (that is, have the general benefits of purely functional code).

See <https://effect.readthedocs.org/> for documentation.

```
class effect.Effect(intent, callbacks=NOTHING)
```

Bases: object

Take an object that describes a desired effect (called an “Intent”), and allow binding callbacks to be called with the result of the effect.

Effects can be performed with `perform()`.

Parameters `intent` – The intent to be performed.

`callbacks = Attribute(name='callbacks', default=Factory(factory=<type 'list'>), validator=None, repr=True, cmp=True)`

`intent = Attribute(name='intent', default=NOTHING, validator=None, repr=True, cmp=True, hash=True, init=True)`

`on (success=None, error=None)`

Return a new Effect with the given success and/or error callbacks bound.

The result of the Effect will be passed to the first callback. Any callbacks added afterwards will receive the result of the previous callback. Normal return values are passed on to the next success callback, and exceptions are passed to the next error callback as a `sys.exc_info()` tuple.

If a callback returns an *Effect*, the result of that *Effect* will be passed to the next callback.

```
effect.perform(dispatcher, effect)
```

Perform an effect and invoke callbacks bound to it.

The dispatcher will be called with the intent, and is expected to return a performer (another callable). See *TypeDispatcher* and *ComposedDispatcher* for some implementations of dispatchers, and `effect.base_dispatcher` for a dispatcher supporting basic intents like *Constant* et al.

The performer will often be decorated with `sync_performer()` or the `deferred_performer` from *txeffect* and will be invoked with the dispatcher¹ and the intent, and should perform the desired effect.² The performer should return the result of the effect, or raise an exception, and the result will be passed on to the first callback, then the result of the first callback will be passed to the next callback, and so on.

Both performers and callbacks may return regular values, raise exceptions, or return another Effect, which will be recursively performed, such that the result of the returned Effect becomes the result passed to the next callback. In the case of exceptions, the next error-callback will be called with a `sys.exc_info()`-style tuple.

Note that this function does *not* return the final result of the effect. You may instead want to use `sync_perform()` or `effect.twisted.perform()`.

Returns None

```
exception effect.NoPerformerFoundError
```

Bases: exceptions.Exception

Raised when a performer for an intent couldn't be found.

```
exception effect.NotSynchronousError
```

Bases: exceptions.Exception

Performing an effect did not immediately return a value.

```
effect.sync_perform(dispatcher, effect)
```

Perform an effect, and return its ultimate result. If the final result is an error, the exception will be raised.

¹ The dispatcher is passed because some performers need to make recursive calls to `perform()`, because they need to perform other effects (see `parallel()` and `perform_parallel_async()` for an example of this).

² Without using one of those decorators, the performer is actually passed three arguments, not two: the dispatcher, the intent, and a “box”. The box is an object that lets the performer provide the result, optionally asynchronously. To provide the result, use `box.succeed(result)` or `box.fail(exc_info)`, where `exc_info` is a `sys.exc_info()`-style tuple. Decorators like `sync_performer()` simply abstract this away.

This requires that the effect (and all effects returned from any of its callbacks) be synchronous. If the result is not available immediately, `NotSynchronousError` will be raised.

`effect.sync_performer(f)`

A decorator for performers that return a value synchronously.

This decorator should be used if performing the intent will be synchronous, i.e., it will block until the result is available and the result will be simply returned. This is the common case unless you're using an asynchronous framework like Twisted or asyncio.

Note that in addition to returning (or raising) values as normal, you can also return another Effect, in which case that Effect will be immediately performed with the same dispatcher. This is useful if you're implementing one intent which is built on top of other effects, without having to explicitly perform them.

The function being decorated is expected to take a dispatcher and an intent, and should return or raise normally. The wrapper function that this decorator returns will accept a dispatcher, an intent, and a box (conforming to the performer interface). The wrapper deals with putting the return value or exception into the box.

Example:

```
@sync_performer
def perform_foo(dispatcher, foo):
    return do_side_effect(foo)
```

`class effect.Delay(delay)`

Bases: `object`

An intent which represents a delay in time.

When performed, the specified delay will pass and then the effect will result in `None`.

Parameters `delay` (*float*) – The number of seconds to delay.

`delay = Attribute(name='delay', default=NOTHING, validator=None, repr=True, cmp=True, hash=True, init=True)`

`class effect.ParallelEffects(effects)`

Bases: `object`

An effect intent that asks for a number of effects to be run in parallel, and for their results to be gathered up into a sequence.

`effect.async.perform_parallel_async()` can perform this Intent assuming all child effects have asynchronous performers. `effect.threads.perform_parallel_with_pool()` can perform blocking performers in a thread pool.

Note that any performer for this intent will need to be compatible with performers for all of its child effects' intents. Notably, if child effects have blocking performers, the threaded performer should be used, and if they're asynchronous, the asynchronous performer should be used.

Performers of this intent must fail with a `FirstError` exception when any child effect fails, representing the first error.

Parameters `effects` – Effects to be performed in parallel.

`effects = Attribute(name='effects', default=NOTHING, validator=None, repr=True, cmp=True, hash=True, init=True)`

`effect.parallel(effects)`

Given multiple Effects, return one Effect that represents the aggregate of all of their effects. The result of the aggregate Effect will be a list of their results, in the same order as the input to this function. If any child effect fails, the first such failure will be propagated as a `FirstError` exception. If additional error information is desired, use `parallel_all_errors()`.

This is just a convenience wrapper for returning of Effect of `ParallelEffects`.

Parameters **effects** – Effects which should be performed in parallel.

Returns An Effect that results in a list of results, or which fails with a *FirstError*.

`effect.parallel_all_errors(effects)`

Given multiple Effects, return one Effect that represents the aggregate of all of their effects. The result of the aggregate Effect will be a list of their results, in the same order as the input to this function.

This is like *parallel()*, but it differs in that exceptions from all child effects will be accumulated and provided in the return value, instead of just the first one.

Parameters **effects** – Effects which should be performed in parallel.

Returns An Effect that results in a list of `(is_error, result)` tuples, where `is_error` is True if the child effect raised an exception, in which case `result` will be an `exc_info` tuple. If `is_error` is False, then `result` will just be the result as provided by the child effect.

`class effect.Constant(result)`

Bases: `object`

An intent that returns a pre-specified result when performed.

Parameters **result** – The object which the Effect will result in.

result = `Attribute(name='result', default=NOTHING, validator=None, repr=True, cmp=True, hash=True, init=True)`

`class effect.Error(exception)`

Bases: `object`

An intent that raises a pre-specified exception when performed.

Parameters **exception** (*BaseException*) – Exception instance to raise.

exception = `Attribute(name='exception', default=NOTHING, validator=None, repr=True, cmp=True, hash=True, init=True)`

`exception effect.FirstError(exc_info, index)`

Bases: `exceptions.Exception`

One of the effects in a *ParallelEffects* resulted in an error. This represents the first such error that occurred.

exc_info = `Attribute(name='exc_info', default=NOTHING, validator=None, repr=True, cmp=True, hash=True, init=True)`

index = `Attribute(name='index', default=NOTHING, validator=None, repr=True, cmp=True, hash=True, init=True)`

`class effect.Func(func)`

Bases: `object`

An intent that returns the result of the specified function.

Note that `Func` is something of a cop-out. It doesn't follow the convention of an intent being transparent data that is easy to introspect, since it just wraps an opaque callable. This has two drawbacks:

- it's harder to test, since the only thing you can do is call the function, instead of inspect its data.
- it doesn't offer any ability for changing the way the effect is performed.

If you use `Func` in your application code, know that you are giving up some ease of testing and flexibility. It's preferable to represent your intents as inert objects with public attributes of simple data. However, this is useful for integrating with "legacy" side-effecting code in a quick way.

Parameters **func** – The function to call when this intent is performed.

func = `Attribute(name='func', default=NOTHING, validator=None, repr=True, cmp=True, hash=True, init=True)`

`class effect.TypeDispatcher(mapping)`

Bases: object

An Effect dispatcher which looks up the performer to use by type.

Parameters `mapping` – mapping of intent type to performer

`mapping = Attribute(name='mapping', default=NOTHING, validator=None, repr=True, cmp=True, hash=True, init=True)`

`class effect.ComposedDispatcher(dispatchers)`

Bases: object

A dispatcher which composes other dispatchers.

The dispatchers given will be searched in order until a performer is found.

Parameters `dispatchers` – Dispatchers to search.

`dispatchers = Attribute(name='dispatchers', default=NOTHING, validator=None, repr=True, cmp=True, hash=True, init=True)`

`effect.catch(exc_type, callable)`

A helper for handling errors of a specific type.

`eff.on(error=catch(SpecificException, lambda exc_info: "got an error!"))`

If any exception other than a `SpecificException` is thrown, it will be ignored by this handler and propagate further down the chain of callbacks.

1.3.2 Submodules

effect.async module

Generic asynchronous performers.

`effect.async.perform_parallel_async(dispatcher, intent, box)`

A performer for `ParallelEffects` which works if all child Effects are already asynchronous. Use this for things like Twisted, asyncio, etc.

WARNING: If this is used when child Effects have blocking performers, it will run them in serial, not parallel.

effect.do module

An imperative-looking notation for Effectful code.

See `do()`.

`effect.do.do(f)`

A decorator which allows you to use `do` notation in your functions, for imperative-looking code:

```
@do
def foo():
    thing = yield Effect(Constant(1))
    yield do_return('the result was %r' % (thing,))

eff = foo()
return eff.on(...)
```

`@do` must decorate a generator function (not any other type of iterator). Any yielded values must either be Effects or the result of a `do_return()` call. The result of a yielded Effect will be passed back into the generator as the result of the `yield` expression. Yielded `do_return()` values will provide the ultimate

result of the Effect that is returned by the decorated function. Note that `do_return()` is only necessary for Python 2 compatibility; return statements can be used directly in Python 3-only code.

It's important to note that any generator function decorated by `@do` will no longer return a generator, but instead it will return an Effect, which must be used just like any other Effect.

Errors are also converted to normal exceptions:

```
@do
def foo():
    try:
        thing = yield Effect(Error(RuntimeError('foo')))
    except RuntimeError:
        yield do_return('got a RuntimeError as expected')
```

(This decorator is named for Haskell's `do` notation, which is similar in spirit).

`effect.do.do_return(val)`

Specify a return value for a `@do` function.

The result of this function must be yielded. e.g.:

```
@do
def foo():
    yield do_return('hello')
```

If you're writing Python 3-only code, you don't need to use this function, and can just use the `return` statement as normal.

effect.fold module

exception `effect.fold.FoldError(accumulator, wrapped_exception)`

Bases: `exceptions.Exception`

Raised when one of the Effects passed to `fold_effect()` fails.

Variables

- **accumulator** – The data accumulated so far, before the failing Effect.
- **wrapped_exception** – The `exc_info` tuple representing the original exception raised by the failing Effect.

`effect.fold.fold_effect(f, initial, effects)`

Fold over the results of effects, left-to-right.

This is like `functools.reduce()`, but instead of acting on plain values, it acts on the results of effects.

The function `f` will be called with the accumulator (starting with `initial`) and a result of an effect repeatedly for each effect. The result of the previous call will be passed as the accumulator to the next call.

For example, the following code evaluates to an Effect of 6:

```
fold_effect(operator.add, 0, [Effect(Constant(1)),
                             Effect(Constant(2)),
                             Effect(Constant(3))])
```

If no elements were in the list, Effect would result in 0.

Parameters

- **f (callable)** – function of (accumulator, element) -> accumulator

- **initial** – The value to be passed as the accumulator to the first invocation of `f`.
- **effects** – sequence of Effects.

`effect.fold.sequence` (*effects*)

Perform each Effect serially, collecting their results into a list.

Raises `FoldError` with the list accumulated so far when an effect fails.

effect.ref module

class `effect.ref.Reference` (*initial*)

Bases: `object`

An effectful mutable variable, suitable for sharing between multiple logical threads of execution, that can be read and modified in a purely functional way.

Compare to Haskell's `IORef` or Clojure's `atom`.

Note Warning: Instantiating a `Reference` causes an implicit side-effect. In other words, `Reference` is not a referentially transparent function, and you can't use equational reasoning on it: a call to `Reference` is not interchangeable with the *result of* a call to `Reference`, since identity matters. If you want to create references in purely functional code, you can use the `effect.Func` intent: `effect.Effect(effect.Func(Reference))`.

modify (*transformer*)

Return an Effect that updates the value with `fn(old_value)`.

Parameters `transformer` – Function that takes old value and returns the new value.

This is not guaranteed to be linearizable if multiple threads are modifying the reference at the same time. It is safe to assume consistent modification as long as you're not using multiple threads, though.

read ()

Return an Effect that results in the current value.

class `effect.ref.ReadReference` (*ref*)

Bases: `object`

Intent that gets a `Reference`'s current value.

ref = `Attribute(name='ref', default=NOTHING, validator=None, repr=True, cmp=True, hash=True, init=True)`

class `effect.ref.ModifyReference` (*ref, transformer*)

Bases: `object`

Intent that modifies a `Reference` value in-place with a transformer func.

This intent is not necessarily linearizable if multiple threads are modifying the same reference at the same time.

ref = `Attribute(name='ref', default=NOTHING, validator=None, repr=True, cmp=True, hash=True, init=True)`

transformer = `Attribute(name='transformer', default=NOTHING, validator=None, repr=True, cmp=True, hash=True)`

`effect.ref.perform_read_reference` (**args, **kwargs*)

Performer for `ReadReference`.

`effect.ref.perform_modify_reference` (**args, **kwargs*)

Performer for `ModifyReference`.

This performer is not linearizable if multiple physical threads are modifying the same reference at the same time.

effect.retry module

Retrying effects.

`effect.retry.retry(effect, should_retry)`

Retry an effect as long as it raises an exception and as long as the `should_retry` error handler returns an Effect of True.

If `should_retry` returns an Effect of False, then the returned effect will fail with the most recent error from `func`.

Parameters

- **effect** (`effect.Effect`) – Any effect.
- **should_retry** – A function which should take an `exc_info` tuple as an argument and return an effect of bool.

effect.testing module

Various functions and dispatchers for testing effects.

Usually the best way to test effects is by using `perform_sequence()`.

`effect.testing.perform_sequence(seq, eff, fallback_dispatcher=None)`

Perform an Effect by looking up performers for intents in an ordered “plan”.

First, an example:

```
@do
def code_under_test():
    r = yield Effect(MyIntent('a'))
    r2 = yield Effect(OtherIntent('b'))
    yield do_return((r, r2))

def test_code():
    seq = [
        (MyIntent('a'), lambda i: 'result1'),
        (OtherIntent('b'), lambda i: 'result2')
    ]
    eff = code_under_test()
    assert perform_sequence(seq, eff) == ('result1', 'result2')
```

Every time an intent is to be performed, it is checked against the next item in the sequence, and the associated function is used to calculate its result. Note that the objects used for intents must provide a meaningful `__eq__` implementation, since they will be checked for equality. Using something like `attrs` or `pyrsistent`’s `PClass` is recommended for your intents, since they will auto-generate `__eq__` and many other methods useful for immutable objects.

If an intent can’t be found in the sequence or the fallback dispatcher, an `AssertionError` is raised with a log of all intents that were performed so far. Each item in the log starts with one of three prefixes:

- sequence: this intent was found in the sequence
- fallback: a performer for this intent was provided by the fallback dispatcher
- NOT FOUND: no performer for this intent was found.

Parameters

- **sequence** (*list*) – List of (*intent*, *fn*) tuples, where *fn* is a function that should accept an intent and return a result.
- **eff** (*Effect*) – The Effect to perform.
- **fallback_dispatcher** – A dispatcher to use for intents that aren't found in the sequence. if None is provided, `base_dispatcher` is used.

class `effect.testing.SequenceDispatcher` (*sequence*)

Bases: `object`

A dispatcher which steps through a sequence of (*intent*, *func*) tuples and runs *func* to perform intents in strict sequence.

This is the dispatcher used by `perform_sequence()`. In general that function should be used directly, instead of this dispatcher.

It's important to use `with sequence.consume()`: to ensure that all of the intents are performed. Otherwise, if your code has a bug that causes it to return before all effects are performed, your test may not fail.

None is returned if the next intent in the sequence is not equal to the intent being performed, or if there are no more items left in the sequence (this is standard behavior for dispatchers that don't handle an intent). This lets this dispatcher be composed easily with others.

Parameters *sequence* (*list*) – Sequence of (*intent*, *fn*).

consume (**args*, ***kws*)

Return a context manager that can be used with the *with* syntax to ensure that all steps are performed by the end.

consumed ()

Return True if all of the steps were performed.

sequence = `Attribute(name='sequence', default=NOTHING, validator=None, repr=True, cmp=True, hash=True, init=True)`

class `effect.testing.EQDispatcher` (*mapping*)

Bases: `object`

An equality-based (constant) dispatcher.

This dispatcher looks up intents by equality and performs them by returning an associated constant value.

This is sometimes useful, but `perform_sequence()` should be preferred, since it constrains the order of effects, which is usually important.

Users provide a mapping of intents to results, where the intents are matched against the intents being performed with a simple equality check (not a type check!).

The mapping must be provided as a sequence of two-tuples. We don't use a dict because we don't want to require that the intents be hashable (in practice a lot of them aren't, and it's a pain to require it). If you want to construct your mapping as a dict, you can, just pass in the result of `d.items()`.

e.g.:

```
>>> sync_perform(EQDispatcher([(MyIntent(1, 2), 'the-result')]),
...               Effect(MyIntent(1, 2)))
'the-result'
```

assuming `MyIntent` supports `__eq__` by value.

Parameters *mapping* (*list*) – A sequence of tuples of (*intent*, *result*).

mapping = `Attribute(name='mapping', default=NOTHING, validator=None, repr=True, cmp=True, hash=True, init=True)`

```
class effect.testing.EQFDispatcher(mapping)
```

Bases: object

An Equality-based function dispatcher.

This dispatcher looks up intents by equality and performs them by invoking an associated function.

This is sometimes useful, but `perform_sequence()` should be preferred, since it constrains the order of effects, which is usually important.

Users provide a mapping of intents to functions, where the intents are matched against the intents being performed with a simple equality check (not a type check!). The functions in the mapping will be passed only the intent and are expected to return the result or raise an exception.

The mapping must be provided as a sequence of two-tuples. We don't use a dict because we don't want to require that the intents be hashable (in practice a lot of them aren't, and it's a pain to require it). If you want to construct your mapping as a dict, you can, just pass in the result of `d.items()`.

e.g.:

```
>>> sync_perform(
...     EQFDispatcher([ (
...         MyIntent(1, 2), lambda i: 'the-result' ) ]),
...     Effect(MyIntent(1, 2)))
'the-result'
```

assuming `MyIntent` supports `__eq__` by value.

Parameters `mapping` (*list*) – A sequence of two-tuples of (intent, function).

mapping = Attribute(name='mapping', default=NOTHING, validator=None, repr=True, cmp=True, hash=True, init=True)

```
effect.testing.resolve_effect(effect, result, is_error=False)
```

Supply a result for an effect, allowing its callbacks to run.

Note that is a pretty low-level testing utility; it's much better to use a higher-level tool like `perform_sequence()` in your tests.

The return value of the last callback is returned, unless any callback returns another Effect, in which case an Effect representing that operation plus the remaining callbacks will be returned.

This allows you to test your code in a somewhat “channel”-oriented way:

```
eff = do_thing() next_eff = resolve_effect(eff, first_result) next_eff = resolve_effect(next_eff, second_result) result = resolve_effect(next_eff, third_result)
```

Equivalently, if you don't care about intermediate results:

```
result = resolve_effect(
    resolve_effect(
        resolve_effect(
            do_thing(),
            first_result),
        second_result),
    third_result)
```

NOTE: parallel effects have no special support. They can be resolved with a sequence, and if they're returned from another effect's callback they will be returned just like any other effect.

Parameters

- **is_error** (*bool*) – Indicate whether the result should be treated as an exception or a regular result.

- **result** – If `is_error` is `False`, this can be any object and will be treated as the result of the effect. If `is_error` is `True`, this must be a three-tuple in the style of `sys.exc_info`.

`effect.testing.fail_effect(effect, exception)`

Resolve an effect with an exception, so its error handler will be run.

class `effect.testing.Stub(intent)`

Bases: `object`

DEPRECATED in favor of using `perform_sequence()`.

An intent which wraps another intent, to flag that the intent should be automatically resolved by `resolve_stub()`.

`Stub` is intentionally not performable by any default mechanism.

intent = `Attribute(name='intent', default=NOTHING, validator=None, repr=True, cmp=True, hash=True, init=True)`

`effect.testing.ESTConstant(x)`

DEPRECATED. Return `Effect(Stub(Constant(x)))`

`effect.testing.ESError(x)`

DEPRECATED. Return `Effect(Stub(Error(x)))`

`effect.testing.ESFunc(x)`

DEPRECATED. Return `Effect(Stub(Func(x)))`

`effect.testing.resolve_stubs(dispatcher, effect)`

DEPRECATED in favor of using `perform_sequence()`.

Successively performs effects with `resolve_stub` until a non-Effect value, or an Effect with a non-stub intent is returned, and return that value.

Parallel effects are supported by recursively invoking `resolve_stubs` on the child effects, if all of their children are stubs.

`effect.testing.resolve_stub(dispatcher, effect)`

DEPRECATED in favor of `perform_sequence()`.

Automatically perform an effect, if its intent is a `Stub`.

Note that `resolve_stubs` is preferred to this function, since it handles chains of stub effects.

effect.threads module

`effect.threads.perform_parallel_with_pool(*args, **kwargs)`

A performer for `effect.ParallelEffects` which uses a `multiprocessing.pool.ThreadPool` to perform the child effects in parallel.

Note that this *can't* be used with a `multiprocessing.Pool`, since you can't pass closures to its `map` method.

This function takes the pool as its first argument, so you'll need to partially apply it when registering it in your dispatcher, like so:

```
my_pool = ThreadPool()
parallel_performer = functools.partial(
    perform_parallel_effects_with_pool, my_pool)
dispatcher = TypeDispatcher({ParallelEffects: parallel_performer, ...})
```

NOTE: `ThreadPool` was broken in Python 3.4.0, but fixed by 3.4.1. This performer should work for any version of Python supported by Effect other than 3.4.0.

Indices and tables

- `genindex`
- `modindex`

e

`effect`, [5](#)
`effect.async`, [9](#)
`effect.do`, [9](#)
`effect.fold`, [10](#)
`effect.ref`, [11](#)
`effect.retry`, [12](#)
`effect.testing`, [12](#)
`effect.threads`, [15](#)

C

callbacks (effect.Effect attribute), 6
catch() (in module effect), 9
ComposedDispatcher (class in effect), 9
Constant (class in effect), 8
consume() (effect.testing.SequenceDispatcher method), 13
consumed() (effect.testing.SequenceDispatcher method), 13

D

Delay (class in effect), 7
delay (effect.Delay attribute), 7
dispatchers (effect.ComposedDispatcher attribute), 9
do() (in module effect.do), 9
do_return() (in module effect.do), 10

E

Effect (class in effect), 5
effect (module), 5
effect.async (module), 9
effect.do (module), 9
effect.fold (module), 10
effect.ref (module), 11
effect.retry (module), 12
effect.testing (module), 12
effect.threads (module), 15
effects (effect.ParallelEffects attribute), 7
EQDispatcher (class in effect.testing), 13
EQFDispatcher (class in effect.testing), 13
Error (class in effect), 8
ESConstant() (in module effect.testing), 15
ESError() (in module effect.testing), 15
ESFunc() (in module effect.testing), 15
exc_info (effect.FirstError attribute), 8
exception (effect.Error attribute), 8

F

fail_effect() (in module effect.testing), 15
FirstError, 8

fold_effect() (in module effect.fold), 10
FoldError, 10
Func (class in effect), 8
func (effect.Func attribute), 8

I

index (effect.FirstError attribute), 8
intent (effect.Effect attribute), 6
intent (effect.testing.Stub attribute), 15

M

mapping (effect.testing.EQDispatcher attribute), 13
mapping (effect.testing.EQFDispatcher attribute), 14
mapping (effect.TypeDispatcher attribute), 9
modify() (effect.ref.Reference method), 11
ModifyReference (class in effect.ref), 11

N

NoPerformerFoundError, 6
NotSynchronousError, 6

O

on() (effect.Effect method), 6

P

parallel() (in module effect), 7
parallel_all_errors() (in module effect), 8
ParallelEffects (class in effect), 7
perform() (in module effect), 6
perform_modify_reference() (in module effect.ref), 11
perform_parallel_async() (in module effect.async), 9
perform_parallel_with_pool() (in module effect.threads), 15
perform_read_reference() (in module effect.ref), 11
perform_sequence() (in module effect.testing), 12

R

read() (effect.ref.Reference method), 11
ReadReference (class in effect.ref), 11
ref (effect.ref.ModifyReference attribute), 11

`ref` (`effect.ref.ReadReference` attribute), 11
`Reference` (class in `effect.ref`), 11
`resolve_effect()` (in module `effect.testing`), 14
`resolve_stub()` (in module `effect.testing`), 15
`resolve_stubs()` (in module `effect.testing`), 15
`result` (`effect.Constant` attribute), 8
`retry()` (in module `effect.retry`), 12

S

`sequence` (`effect.testing.SequenceDispatcher` attribute), 13
`sequence()` (in module `effect.fold`), 11
`SequenceDispatcher` (class in `effect.testing`), 13
`Stub` (class in `effect.testing`), 15
`sync_perform()` (in module `effect`), 6
`sync_performer()` (in module `effect`), 7

T

`transformer` (`effect.ref.ModifyReference` attribute), 11
`TypeDispatcher` (class in `effect`), 8