**IO:**

A data type for encoding side effects as pure values, capable of expressing both synchronous and asynchronous computations.

**BASICS:**

A value of type IO[A] is a computation which, when evaluated, can perform effects before returning a value of type A.

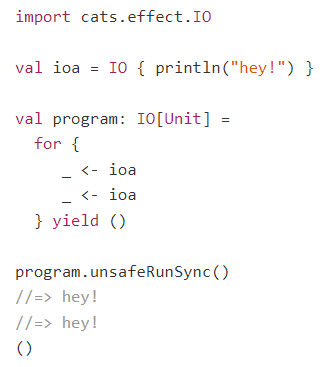
IO values are pure, immutable values and thus preserves referential transparency, being usable in functional programming. An IO is a data structure that represents just a description of a side effectful computation.

IO can describe synchronous or asynchronous computations that:

* on evaluation yield exactly one result
* can end in either Success, Errored or Canceled and in case of failure (Errored or Canceled) flatMap chains get short-circuited (IO implementing the algebra of MonadError)
* can be canceled, but note this capability relies on the user to provide cancellation logic

Effects described via this abstraction are not evaluated until an "unsafe" method is used. Effectful results are not memoized, meaning that memory overhead is minimal (and no leaks), and also that a single effect may be run multiple times in a referentially-transparent manner.

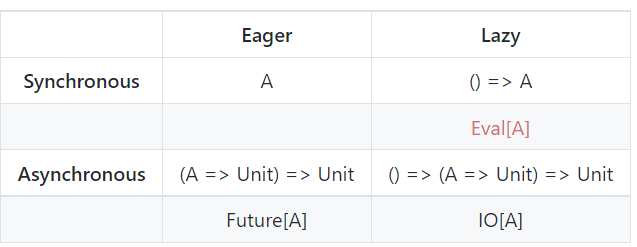
For example:



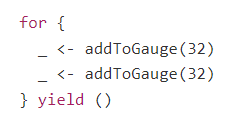
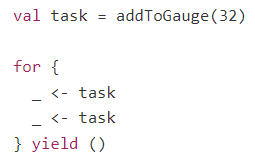
The above example prints "hey!" twice, as the effect re-runs each time it is sequenced in the monadic chain.

**REFERENTIAL TRANSPARENCY AND LAZY EVALUATION:**

IO preserves referential transparency even when dealing with side effects and is lazily evaluated.  
The evaluation model of different types:

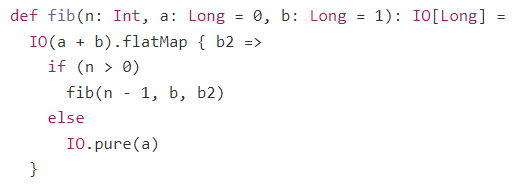


Note: laziness goes hand in hand with referential transparency.  
For example, having referential transparency allows these two examples to give the same result:

**STACK SAFETY:**

IO is trampolined in its flatMap evaluation. This means that you can safely call flatMap in a recursive function, without getting a stack overflow:



IO implements all the type-classes in the cats-effects hierarchy. Therefore, all those operations are available for IO, in addition to some others.

**DESCRIBING EFFECTS**

IO is an abstraction that can efficiently describe multiple kinds of effects:

**Pure Values — IO.pure:**

Lifts pure values into IO, yielding IO values that are "already evaluated" (eagerly evaluated), the following function being defined on IO's companion:



Note: the parameter in pure() is passed by value, not by name.

*pure()* can lift a number (pure value) into IO. Can then compose it with another IO that wraps a side-effect in a safe manner, as nothing is going to be executed:



However, *pure()* can’t suspend side-effects as *pure()* is eagerly evaluated.  
In the case below, the *println* will trigger a side effect (before the IO is run) that is not suspended in IO.



**Synchronous Effects — IO.apply:**

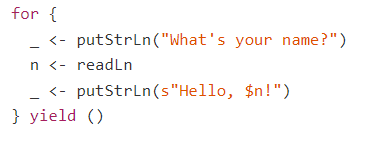
*IO.apply()* is simpler syntax for *Sync[IO].delay*. It describes IO operations that can be evaluated immediately, on the current thread and call-stack:



Note: the parameter is passed ''by name'', its execution being "suspended" in the IO context (Lazily evaluated).

An example would be reading / writing from / to the console, which on top of the JVM uses blocking I/O, so their execution is immediate:





**Asynchronous Effects — IO.async & IO.cancelable:**

Describes asynchronous processes.

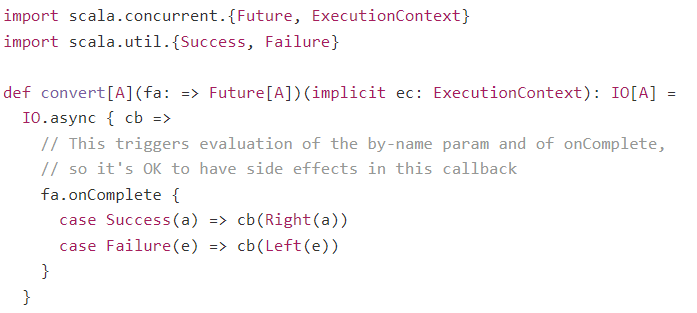
**IO.async:**

*IO.async* is the operation that complies with the laws of Async#async and can describe simple asynchronous processes that cannot be canceled, its signature being:



This injects a callback that you can use to signal either successful results (with Right(a)), or failures (with Left(error)). Users can trigger whatever asynchronous side effects are required, then use the injected callback to signal completion.

For example, you don't need to convert Scala's Future, because you already have a conversion operation defined in IO.fromFuture, however the code for converting a Future would be:



**IO.cancelable:**

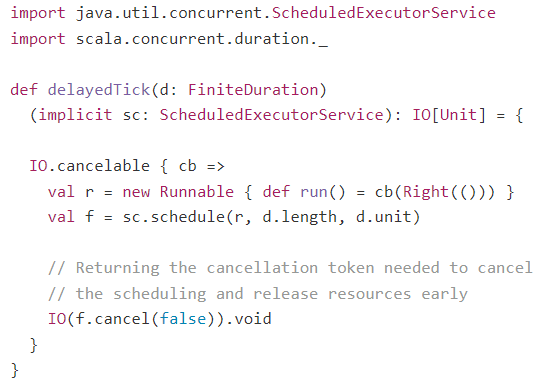
Use *IO.cancelable* to build cancelable IO tasks, whilst being compliant with Concurrent#cancelable:



It is similar to *IO.async*, but in that registration function the user is expected to provide an IO[Unit] that captures the required cancellation logic.

Important: *cancellation is the ability to interrupt an IO task before completion, possibly releasing any acquired resources, useful in race conditions to prevent leaks.*

Example: describe a sleep operation that depends on Java's ScheduledExecutorService, delaying a tick for a certain time duration:



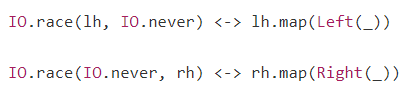
Note: this delayed tick is already described by IO.sleep (via Timer).

**IO.never:**

Represents a non-terminating IO defined in terms of async, useful as shortcut and as a reusable reference:



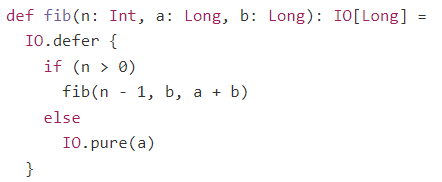
This is useful in order to use non-termination in certain cases, like race conditions.  
For example, given IO.race, we have these equivalences (**<->** means equivalence):



**Deferred Execution — IO.defer:**

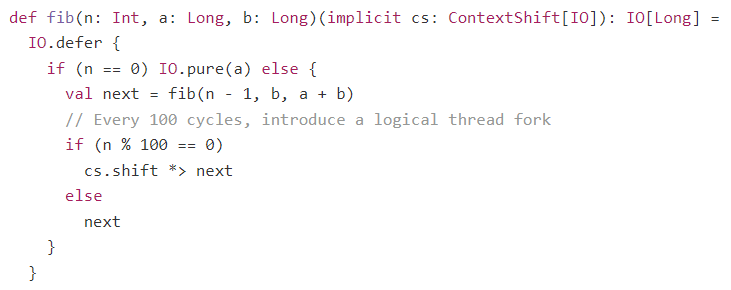
Has this equivalence  


It is useful for suspending effects, but that defers the completion of the returned IO to some other reference. It's also useful for modelling stack safe, tail recursive loops:



By using *IO.defer*, this function’s evaluation is lazy and it's going to use constant memory (so is stack-safe). This also works with flatMap.

Could describe this function using Scala's @tailrec mechanism, however by using IO we can also preserve fairness by inserting asynchronous boundaries:

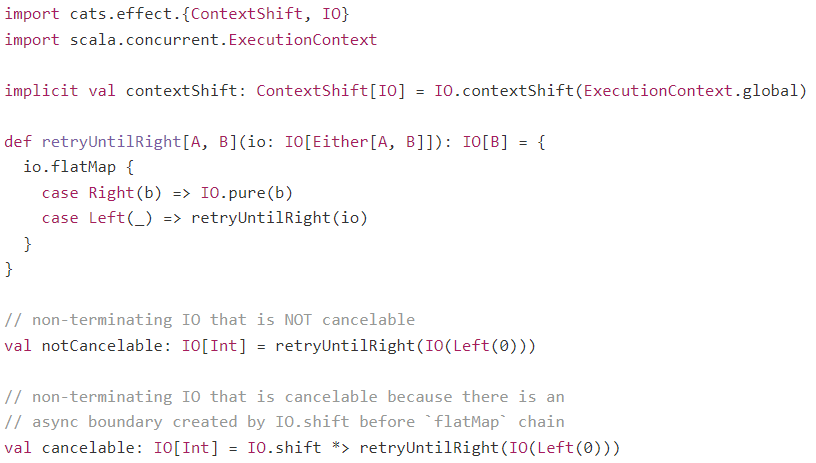


**CONCURRENCY AND CANCELLATION**

IO can describe interruptible asynchronous processes. As an implementation detail:

1. Not all IO tasks are **cancelable**  
   Cancellation status is only checked after asynchronous boundaries. It can be achieved in the following way:
   * Building it with IO.cancelable, IO.async, IO.asyncF or IO.bracket
   * Using IO.cancelBoundary or IO.shift

Note: flatMap chains cancelable only if the chain happens after an asynchronous boundary. After  
an asynchronous boundary, cancellation checks are performed on every N flatMap.  
The value of N is hardcoded to 512. An example:



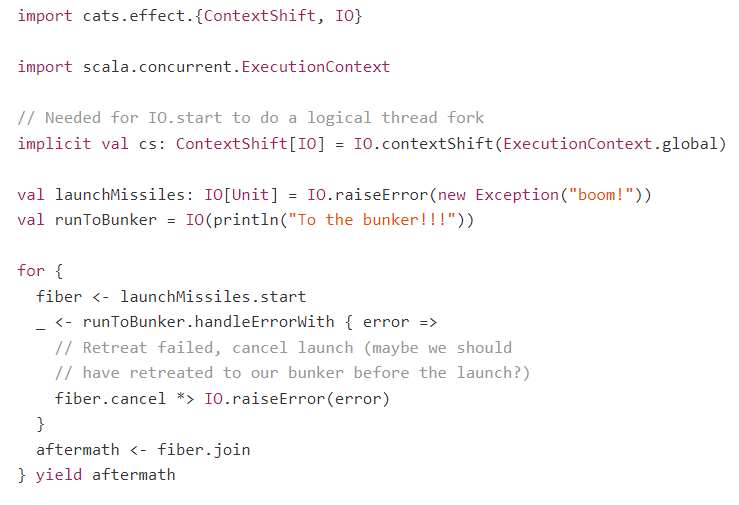
1. IO tasks that are cancelable, usually become non-terminating on cancel.

**STARTING + CANCELING FIBERS:**

To start a fiber, use the IO.start method (which is compliant with Concurrent#start)



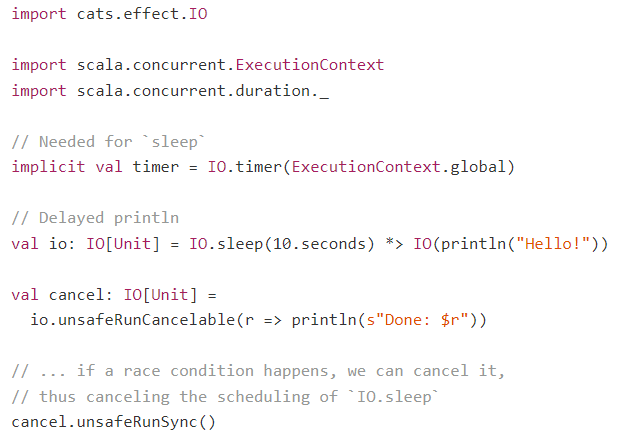
This returns a Fiber, which can either be joined (via *join*) or interrupted (via *cancel*).



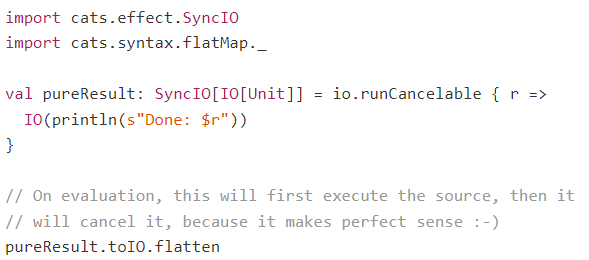
**runCancelable & unsafeRunCancelable:**

The above is the pure cancel method, accessible via Fiber.  
An alternative way to cancel and interrupt tasks is via *runCancelable* (the pure version) and *unsafeRunCancelable* (the unsafe version).

Example: Using unsafeRunCancelable, which is side-effecting and impure:



Example: Using *runCancelable*, which is compliant with the laws of ConcurrentEffect (the difference in implementation is that the actual execution is suspended in SyncIO):

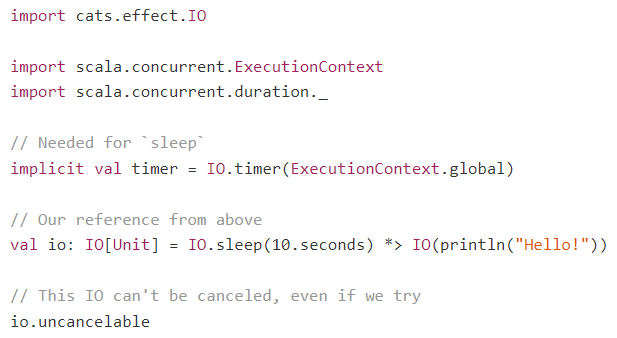


**IO.uncancelable:**

Given a cancelable IO, we can turn it into an IO that cannot be canceled.

To ensure an IO's execution is atomic (either all of it executes or none of it). Cancelable IOs are by definition not atomic and in certain cases we need to make them atomic.

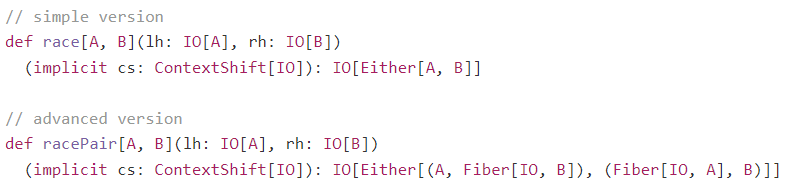
This law is compliant with the laws of Concurrent#uncancelable.



**Race Conditions — race & racePair:**

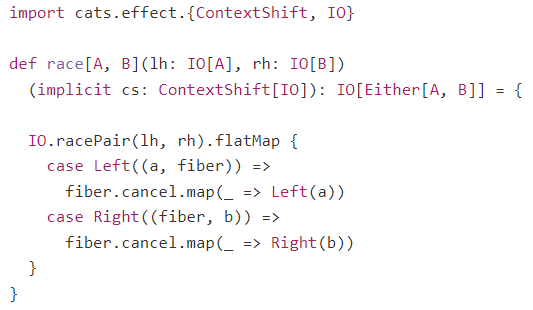
A race condition is a piece of logic that creates a race between two or more tasks, with the winner being signalled immediately, with the losers usually being canceled.

IO provides two operations for races in its companion:



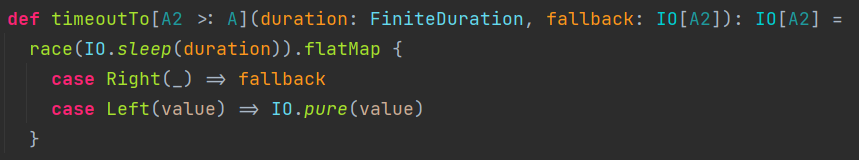
*IO.race* will cancel the loser immediately.  
*IO.racePair* gives you the Fiber of the lose, letting you decide what to do next.

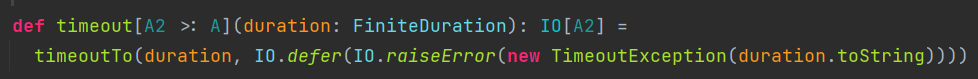
So, *race* can be derived with *racePair* like so:



*IO.timeoutTo* & *IO.timeout* use *race* in their implementations

* *IO.timeoutTo*:
  + Takes an IO to run, a time duraction and a fallback IO
  + Runs the IO. If this IO finishes within the time duration, the result of this IO is returned. Otherwise if the time-duration finishes before the IO finishes, then this IO is canceled and the fallback IO is returned.
* IO.timeout:
  + Same as timeoutTo, but instead of a fallback IO, it throws a TimeoutException if the IO to run loses the race.





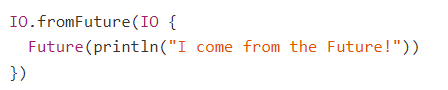
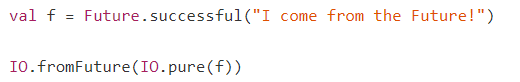
**CONVERSIONS**

**fromFuture (Future -> IO):**

Lifts a Future to an IO, producing either a result or a failure. It is defined as follows:



Future eagerly evaluates and memorizes its result, so this function takes its parameter as an IO, which could be lazily evaluated. If this laziness is appropriately threaded back to the definition site of the Future, it ensures that the computation is fully managed by IO and thus referentially transparent.

* Lazy evaluation:  
  
* Eager evaluation:  
  

**fromEither (Either -> IO):**

Lifts an *Either[Throwable, A]* into the *IO[A]* context raising the throwable if it exists.



**ERROR HANDLING**

All error handling for IO is through MonadError[IO, Throwable], so get all operations available for MonadError (where the error type is a Throwable).

*import cats.syntax.all.\_.*

**raiseError:**

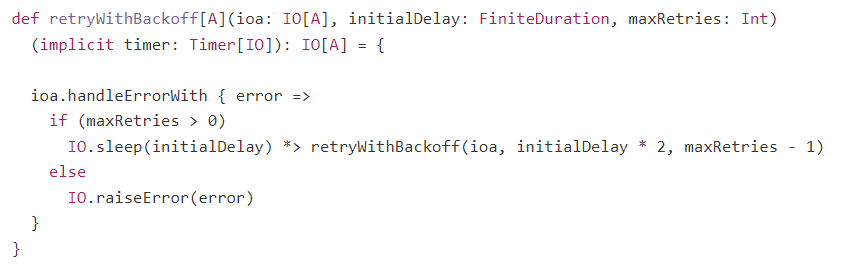
Constructs an IO which sequences the specified exception.  


**attempt:**

Materializes any sequenced exceptions into value space, where they may be handled.  
This is analogous to the catch clause in try/catch, being the inverse of *IO.raiseError*.



Example:  
With IO, model a loop that retries evaluation until success or some other condition is met

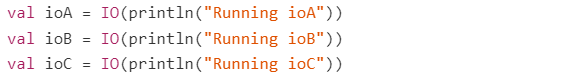
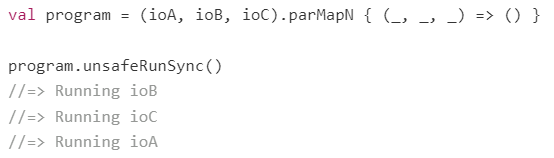


**PARALLELISM**

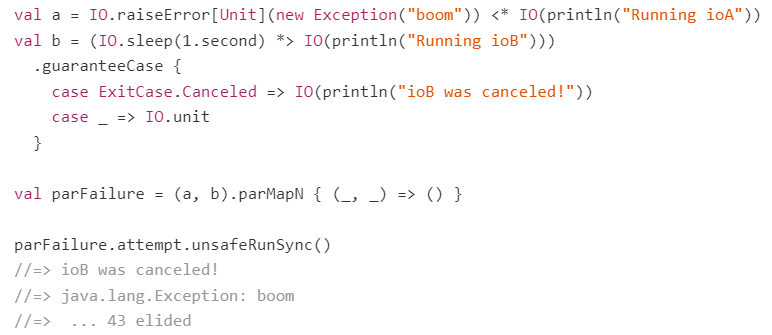
These are higher-level methods that run multiple IOs in parallel.

**parMapN:**

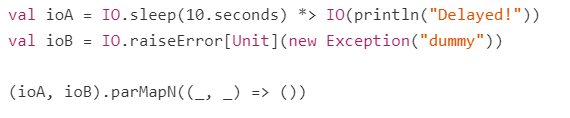
Run an arbitrary number of IOs in parallel, and applies the results of these IOs to a function (this function will accept all the IOs in parMapN in a single function call).  
It finishes processing when all the IOs are completed, either successfully or with a failure. For example:

If any of the IOs completes with a failure then the result of the whole computation will be failed, while the unfinished tasks get cancelled. Example:



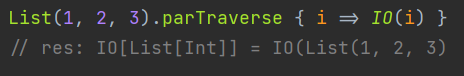
If one of the tasks fails immediately, then the other gets canceled and the computation completes immediately, so in this example the pairing via parMapN will not wait for 10 seconds before emitting the error:



**parTraverse:**

Given a list of items (non-IOs) and a function to turn a single item into an IO, applies this function to each item in the list (in parallel) and returns this list.

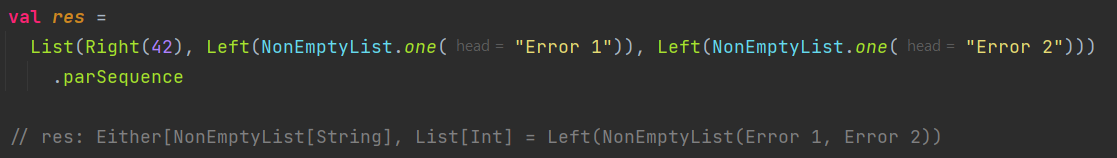
Alternatively, *cats.Traverse.traverse* runs synchronously



**parSequence:**

Given a list of IOs, returns a single IO with the result list. Executes the IO tasks in parallel.  
Compared to *parTraverse*, *parSequence* flattens the list/sequence of IOs.

Alternatively, *cats.Traverse.sequence* runs synchronously



**“UNSAFE” OPERATIONS:**

All operations with prefix “unsafe” are impure functions and perform side effects.  
The structure of your program should be in a monadic way (using functions such as map and flatMap) to compose other functions and have a call to an unsafe operation only once at the end of the program.

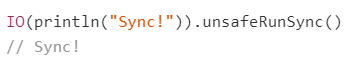
The IO at the end (before being run) will be pure and will just contain the ‘instructions’ of the actions it will run. Then when this IO is run, it will be impure and actually produce side-effects.

**unsafeRunSync:**

Produces the result by running the encapsulated effects as impure side effects.

If any component of the computation is asynchronous, the current thread will block awaiting the results of the async computation. By default, this blocking will be unbounded. To limit the thread block to some fixed time, use *unsafeRunTimed* instead.

Any exceptions raised within the effect will be re-thrown during evaluation.



**unsafeRunAsync:**

Passes the result of the encapsulated effects to the given callback by running them as impure side effects.

Any exceptions raised within the effect will be passed to the callback in the Either. The callback will be invoked at most once. Note that it is very possible to construct an IO which never returns while still never blocking a thread, and attempting to evaluate that IO with this method will result in a situation where the callback is never invoked.



**unsafeRunCancelable:**

Evaluates the source IO, passing the result of the encapsulated effects to the given callback.  
Note that this has the potential to be interrupted.



**unsafeRunTimed:**

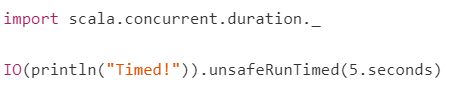
Similar to *unsafeRunSync*, except with a bounded blocking duration when awaiting asynchronous results.

The limit parameter does not limit the time of the total computation, but acts as an upper bound on any individual asynchronous block. Thus, if a 5 second limit is passed:

* To an IO consisting solely of synchronous actions -> the evaluation may take considerably longer than 5 seconds.
* To an IO consisting of several asynchronous actions joined together -> evaluation may take up to n \* 5 seconds, where n is the number of joined async actions.

As soon as an async blocking limit is hit, evaluation "immediately" aborts and None is returned.

Note: This function is intended for testing purposes; it should never appear in your mainline production code! It is absolutely not an appropriate function to use if you want to implement timeouts etc.



**unsafeToFuture:**

Evaluates the effect and produces the result in a Future.

This is similar to unsafeRunAsync in that it evaluates the IO as a side effect in a non-blocking fashion, but uses a Future rather than an explicit callback.  
This function should really only be used if interoperating with legacy code which uses Scala futures.

