

A Design Space Exploration of Async/Await

GAVIN GRAY✉, SHRIRAM KRISHNAMURTHI, and WILL CRICHTON, Brown University, USA

Many modern programming languages include some form of asynchronous programming. In particular, a growing number now have what we call straight-line asynchrony: attempts to provide asynchronous functions that look similar to synchronous functions, thereby enabling asynchrony without introducing complex control. These languages often share construct names like "async" and "await," which suggests that they have deep semantic similarities. Yet, a close examination reveals that these languages are quite different along several dimensions, often subtly. These differences have real semantic consequences: similar-looking programs can exhibit divergent behavior, confusing developers and language designers alike.

This paper therefore presents a *design space exploration* of straight-line asynchrony. We dissect several existing languages, and show how no two of them agree as a whole on design decisions that affect the presence and ordering of execution. We articulate a design space with nine dimensions covering the full lifecycle of an asynchronous computation, covering questions such as: What precise guarantees does a language give upon calling an asynchronous function? What happens at the end of a task's life? How can a task handle being cancelled? We explore these questions through concrete examples, informal design discussion, and a formal semantics. Our ultimate goal is to help programmers, language designers, and language theorists all better understand the emerging landscape of straight-line asynchrony.

CCS Concepts: • **Theory of computation** → **Program semantics**; **Concurrency**.

Additional Key Words and Phrases: asynchronous programming, async/await, structured concurrency

ACM Reference Format:
Gavin Gray, Shriram Krishnamurthi, and Will Crichton. 2026. A Design Space Exploration of Async/Await. *Proc. ACM Program. Lang.* 10, OOPSLA2, Article 387 (October 2026), 28 pages. https://doi.org/10.1145/3839519

1 Introduction

Software developers have long struggled to write concurrent and parallel programs that are both correct and efficient. Programming languages have responded to this challenge by gradually lifting elements of concurrency from user-space to the level of language design. Approaches vary greatly, but one core idea shared between most modern languages is *async/await*: marking blocks of code as **`async`**, and using an **`await`** keyword to wait for an async block to finish executing. These features can be found in recent versions of Python, JavaScript, Swift, Rust, C♯, and more.

Our original motivation for studying async/await was to help developers translate their understanding of the concept from one language to another, e.g., from JavaScript to Rust. In theory, both languages contain the same core concept with small differences in the details. We should be able to explain async/await just as one might explain lexically-scoped variables, if-statements, or other standardized language features [25]. In reality, the surface similarities between async/await designs can mask deep semantic divergences. For instance: what's the difference between a coroutine, a future, a promise, and a task? What happens when you cancel a task? When are tasks required to exit, if ever? How do errors propagate across a task boundary, if at all? These questions are further

Authors' Contact Information: Gavin Gray, gavin_gray@brown.edu; Shriram Krishnamurthi, shriram@brown.edu; Will Crichton, Brown University, Providence, USA, will_crichton@brown.edu.



This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2026 Copyright held by the owner/author(s).
ACM 2475-1421/2026/10-ART387
https://doi.org/10.1145/3839519

```
async fn write_to_log():
  print("A")
  // simulate log write
  await sleep(2)
  print("B")

async fn process_await():
  task = spawn write_to_log()
  await sleep(0)
  await task

async fn process_detach():
  task = spawn write_to_log()
```

(a) Two examples of functions that spawn a task to write data to a log, one that waits for the task, and another that doesn't.

```
async fn ex1():
  t = write_to_log()
  print("C")
  await sleep(3)

async fn ex2():
  await process_detach()
  await sleep(1)
  print("C")

async fn ex3():
  await timeout(
    1, process_await)
  print("C")
  await sleep(3)
```

(b) Three example invocations of the processing functions.

	ex1	ex2	ex3
C♯	ACB	AC	ACB
JavaScript	ACB	ACB	ACB
Swift	CAB	AC	AC
Python			
Asyncio	C	AC	AC
Trio	C	ABC	AC
Rust			
Tokio	C	AC	ACB
Smol	C	C	AC

(c) A table of outcomes when running the programs translated from the pseudocode into different languages.

Fig. 1. An example of an asynchronous programming pattern that exhibits different behavior under seven modern async/await languages and runtimes. We discuss the translation from pseudocode to each respective language in Section 3.

complicated by languages with customizable async/await runtimes (e.g., Rust and Python), which give different answers based on the choice of runtime.

As a concrete example, consider the pseudocode in Figure 1, showing two ways to run a background task that writes information to a log. Languages differ with respect to what happens when an async function is called, they differ with tasks at the end of their spawning scope, and languages also differ in terms of when and how tasks are stopped in the event of cancellation. When comparing seven modern approaches to async/await, *no* two of them exhibit the same behavior on three straightforward calling contexts!

Observing this conceptual chaos, we set out to perform a *design space exploration* of async/await in modern languages. The contribution of this paper is the result of that exploration:

(1) **We start by situating the work with a high-level overview of asynchronous programming** (Section 2). We contrast asynchrony against concurrency and parallelism, and introduce common asynchrony mechanisms such as racing and cancellation.
(2) **We describe the core design dimensions of asynchrony in modern languages** (Section 3). Our goal is to capture the most fundamental distinctions between designs, focusing on design choices that cause similar-looking programs to exhibit divergent behavior.
(3) **We provide a precise description of each design dimension using operational semantics** (Section 4). We give a precise model of each language's async design in terms of variants on a core asynchrony calculus with delimited continuations.

2 Core Concepts in Asynchrony

Async/await are just keywords—what is the underlying concept being expressed across these languages? The obvious answer is *asynchronous programming* or, more simply, *asynchrony*. Like the words concurrency and parallelism, asynchrony does not have a definitive meaning. Rather, its meaning is socially constructed out of the types of programs people want to write. Fifteen years ago, asynchronous programming was "characterized by many simultaneously pending reactions to

internal or external events" [44]. Today, we might add to that list: running I/O-bound computations in parallel, or having fine-grained control over the execution of computations. The scope of asynchronous programming has expanded.

Moreover, async/await is only one form of asynchrony. The job of async/await is to avoid the inversion of control associated with callbacks and event loops. The goal, as the C# documentation says, is "to enable code that reads like a sequence of statements, but executes in a more complicated order" [46]. Async/await is about leaning on the language to simplify control-flow patterns, as opposed to implementing asynchrony abstractions entirely at the library level.

Before diving into the details of async design dimensions, we start by providing a high-level picture of async/await circa 2026: how does asynchrony relate to the similar concepts of concurrency and parallelism, and what does it look like to write async/await code today?

2.1 Asynchrony vs. Concurrency vs. Parallelism

We will define concurrency as a logical property of a computation, whereby two subcomputations can execute interleaved: one computation begins before another ends. Parallelism is a physical property of a computation, whereby two subcomputations can execute simultaneously: both computations overlap at the same point in time. Parallelism implies concurrency but not vice versa. For instance, an embedded operating system running on a single core is concurrent but not parallel.

We think of asynchrony as a special case of cooperative concurrency. In an asynchronous program, two subcomputations execute interleaved because they are written to yield control to one another. Asynchrony contrasts with competitive concurrency, where interleaving occurs through external imposition, e.g., by context switches. Competitive concurrency usually arises at an architectural level below the language, such as operating system threads and interrupts.

Concurrency and parallelism have programming paradigms. Paradigms of concurrent computing include shared-memory concurrency (e.g., pthreads), communicating sequential processes (MPI, Go), and actors (Erlang). Paradigms of parallel computing include fork-join parallelism (Cilk, OpenMP), model parallelism (MapReduce, Spark), and data parallelism (OpenGL, CUDA). Like all paradigms, these labels are useful but leaky, such as how a complex parallel program often involves shared-memory concurrency. Ultimately, a paradigm is about focusing a set of related mechanisms (`spawn`, mutexes) on a set of related problems (sorting a list, running a database).

Asynchrony also has paradigms. Callback asynchrony uses higher-order functions to provide a continuation for the completion of a computation, most notably found in C#, JavaScript, and Swift. Event-loop asynchrony structures a program around a main loop that dispatches reactions to an event stream, most notably found in C++ and Rust. Languages have historically developed async/await to address the ergonomic problems associated with these two asynchrony paradigms. A key motivation is to provide a surface syntax for concurrent code that looks comparable to non-concurrent (synchronous) code. For example, below are excerpts from async/await proposal documents across a few languages (see the Supplemental Material for additional examples):

> **(C#)** "Notice first how similar it is to the synchronous code. . . . The control flow is completely unaltered, and there are no callbacks in sight. That doesn't mean that there are no callbacks, but the compiler takes care of creating and signing them up . . . " [45]

> **(Python)** "It is proposed to make coroutines a proper standalone concept in Python, and introduce new supporting syntax. The ultimate goal is to help establish a common, easily approachable, mental model of asynchronous programming in Python and make it as close to synchronous programming as possible." [37]

> **(Rust)** "From a user's perspective, they can use async/await as if it were synchronous code, and only need to annotate their functions and calls." [50]

```javascript
async function timeout<T>(
  secs: number,
  f: () => Promise<T>
): Promise<T | null> {
  return await Promise.race([
    f(), sleep(secs)
  ]);
}
```

(a) JavaScript using its built-in Promise API. `Promise.race` returns the result of whichever future finishes first. JavaScript does not cancel the losing computation.

```python
async def timeout(
  secs: float,
  f: Callable[[],
       Awaitable[T]]
) -> T | None:
  with trio.move_on_after(
    secs
  ):
    return await f()
```

(b) Python+Trio using its first-class construct for timeout. Unlike other languages sampled, Python+Trio does not provide a racing primitive. The timed-out computation is cancelled.

```rust
async fn timeout<F: Future>(
  d: Duration,
  f: impl FnOnce() -> F
) -> Option<F::Output> {
  tokio::select! {
    result = f() =>
      Some(result),
    _ = sleep(d) =>
      None,
  }
}
```

(c) Rust+Tokio using `select!`, which dispatches based on the result of a race. The losing computation is cancelled.

Fig. 2. A simplified implementation of `timeout`, demonstrating primitives for racing. The minimal semantics for timing out does not guarantee that a timed-out computation is cancelled, it simply guarantees that execution is no longer blocked.

2.2 Straight-Line Asynchrony

We therefore define the paradigm embodied by async/await as *straight-line asynchrony*, given its fundamental organizing principle of eliminating complex control flow. An alternative name would be *task asynchrony*, used within the C# community. As we will discuss, tasks are indeed a central concept to async/await, but when looking cross-linguistically, tasks are not the only element of straight-line asynchrony. For instance, languages like Rust provide coroutines so straight-line asynchrony can be achieved without ever creating a task. Some designs like Python+Trio do not expose any concept of a task to the user.

The goal of this paper is to explore the design space of straight-line asynchrony. We primarily focus on programming languages that (a) have a relatively settled design for straight-line asynchrony, and (b) are in relatively widespread use. That includes languages with a singular form of asynchrony: JavaScript (specifically ECMAScript 2025, with TypeScript types for clarity), C# (v14 on .NET 10), and the *structured-concurrency* subset of Swift (v6.2). It also includes languages with a modular form of asynchrony: Python (v3.14, using Asyncio and Trio v0.33) and Rust (v1.92.0, with Tokio v1.50.0 and Smol v2.0.2). We discuss how our findings relate to other languages in Section 6. Throughout, we consider only async function calls that introduce concurrency. We do not consider immediately-awaited applications (e.g., `await f()`), which complete before the caller resumes and therefore behave like synchronous calls under every design we study. We describe our findings informally in Section 3 and formally in Section 4.

We do not expect our readers to be familiar with the concrete syntax of all relevant languages, so we articulate the design dimensions at a high level, and present code examples primarily using the same pseudocode style in Figure 1. However, it is useful to first get a sense for both the gestalt and the variety of syntaxes, mechanisms, and semantics in straight-line asynchrony. We therefore conclude this section by implementing an async utility in multiple languages.

```
async Task<T> Timeout<T>(
  TimeSpan duration,
  CancellationToken token,
  Func<CancellationToken, Task<T>> f
) {
  using var childCts = CancellationTokenSource
    .CreateLinkedTokenSource(token);
  var work = f(childCts.Token);
  try {
    return await work.WaitAsync(
      duration, token);
  } catch (Exception e) when (
    e is TimeoutException or
     OperationCanceledException
  ) {
    childCts.Cancel();
    throw;
  }
}
```

(a) C# does not provide a built-in mechanism to cancel a `Task`, graceful shutdown requires the use of a cancellation token threaded through each async function. The use of a child token, `childCts`, ensures that cancelling `f` does not cancel other tasks observing the `token` input to `Timeout`.

```
func timeout<T: Sendable>(
  duration: Duration,
  f: @Sendable () async throws -> T
) async throws -> T? {
  try await withoutActuallyEscaping(f) {
    f in
    try await withThrowingTaskGroup {
      group in
      group.addTask { try await f() }
      group.addTask {
        try await Task.sleep(for: duration)
        return nil
      }
      defer { group.cancelAll() }
      return try await group.next()!
    }
  }
}
```

(b) Swift. The `TaskGroup` interface automatically enforces that `f` is cancelled if `timeout` is cancelled, and the `defer` block enforces that `f` is cancelled if the sleep returns first. Each block is wrapped in `try` because awaiting a cancelled task can raise an exception.

Fig. 3. Enhanced implementations of `timeout` handling cancellation both from within and without.

2.3 A Case Study on Timing Out

Consider implementing a `timeout` function, which takes as input an asynchronous function and a duration. The minimal semantics is that `timeout` returns either the result of the asynchronous work, or if the work takes longer than the duration, an indication of timing out. The `timeout` function is a paradigmatic example of asynchronous programming, as its implementation is often non-trivial when using only primitives from other concurrency or parallelism paradigms.

Most straight-line async designs include a race primitive (also called `select`, `or`, `wait`, or `next`) that returns the first of several asynchronous computations to complete. Figure 2 shows three implementations of the minimal `timeout` semantics using these primitives. The dual of racing is to wait for all computations to complete, variously called `all`, `and`, `join`, or `gather`.

An enhanced semantics for `timeout` should take into account *cancellation*: stopping a computation that is no longer needed. Cancellation is another core concept for asynchrony, which did not feature prominently in earlier designs such as JavaScript and C#, but is more consciously part of later designs such as Swift and Python. For `timeout`, we desire three properties. First, if `f` exceeds the duration, then `f` is cancelled. Second, if `timeout` is cancelled, then `f` is also cancelled. Third, if `f` is cancelled, then it can gracefully shut down, i.e., execute cleanup code prior to termination.

These goals can be achieved either through library-level or language-level implementations of cancellation. At the library level, a token or signal is threaded through a program, and each task is responsible for observing that token. At the language level, tasks are grouped together, and cancelling one task propagates the cancellation to other tasks, often by throwing a cancellation

exception. Figure 3 shows a library-level and language-level implementation of the enhanced `timeout` semantics.

For the pseudocode in this paper we use the minimal semantics implementation of `timeout` as our cancellation primitive. We made this choice to encompass the varying primitives exposed by runtimes, as well as to accommodate languages without built-in cancellation (e.g., JavaScript, C♯).

In sum, Figure 1 demonstrated how different designs for straight-line asynchrony can take the same simple-looking program and cause different outcomes. This section demonstrated how for more sophisticated asynchronous patterns, the same concept can require substantively different implementations. Together, they motivate the need for a better understanding of the collective design space of straight-line asynchrony.

3 Design Dimensions

Every design for straight-line asynchrony is composed of dozens of individual design decisions that trade off performance, correctness, usability, extensibility, interoperability, and so on. To scope this paper, we focus on design decisions that substantively influence the *functionality* of asynchronous programs. As suggested by Figure 1, we want to explain how similar-looking programs can have divergent behavior in terms of the order or presence of operations.

By contrast, this paper does not attempt to account for design decisions that are principally about *ergonomics*: for example, should **`await`** be a prefix or postfix operator in the concrete syntax? This paper also does not focus on design decisions that are principally about *performance*: for example, how should an asynchronous runtime efficiently interface with the operating system? However, for straight-line asynchrony, performance and functionality are difficult to disentangle. Several design dimensions we discuss will be motivated by performance, but the key criterion for inclusion is their ultimate impact on functionality.

To perform this design space exploration, we manually analyzed the designs of today's languages containing straight-line asynchrony features. We ultimately focused on the languages enumerated in Section 2.2 (JavaScript, C♯, Swift, Python, Rust), but we also examined Kotlin, C++20, F♯, Haskell, OCaml, Hack, Nim, Dart, and Zig. We read design documents (e.g., RFCs, published papers), GitHub discussions, and community blogs. In addition, we drew on reports from community members of confusing or surprising behavior in async programs [2, 24, 32–34, 36, 38, 39, 48].

We identified nine design dimensions, collected into Table 1. These dimensions are grouped into three categories, Start of Life, End of Life, and Cancellation, which serve to organize the remainder of this section. For each design dimension, we explain the points along that dimension, and the design justifications offered by different languages for selecting each point.

3.1 Start of Life

An asynchronous computation is generally defined through a function explicitly marked as asynchronous, such as the **`async fn`** `write_to_log` in Figure 1a. To begin invoking an asynchronous function in any language, one simply calls the function, as in `write_to_log()`. Immediately, the semantics diverge along the first axis: Eagerness.

3.1.1 Eagerness. Consider the program in Figure 4, where an async function `work` is called twice, and later awaited. Based on each language's approach to async function calls, three outcomes can occur: lazy, where no work occurs until being awaited, eager, where work occurs immediately in the main thread of execution, and semi-eager, where no work occurs but is immediately registered in the runtime to start soon, potentially on a different thread, resulting in non-deterministic output. To understand this variation, we first need to discuss two foundational concepts for straight-line asynchrony: coroutines and tasks.

Table 1. Overview of the asynchronous programming design dimensions

Start of Life			
Eagerness How to evaluate an async function application.	LAZY Evaluate to a coroutine without executing further. *Python* · *Rust*	EAGER Evaluate in current thread, and schedule as task on await. *C♯* · *JavaScript* · *Swift* (immediate)	SEMI-EAGER Evaluate to task, and schedule task for execution. *Swift* (async let)
Suspension Guarantees on whether await points suspend.	STATIC Await points guaranteed to suspend. *JavaScript*	DYNAMIC No guarantees on awaiting tasks. *C♯* · *Swift* · *Tokio* · *Smol* · *Asyncio* · *Trio*	
End of Life			
Extent The default interval of time during which a task may exist.	INDEFINITE Tasks by default may exist until the end of the runtime. *JavaScript* · *C♯* · *Tokio* · *Smol* · *Asyncio*	DYNAMIC Tasks by default may exist until the end of their spawning scope. *Swift* · *Trio*	
Reference Strength [For INDEFINITE EXTENT] The type of reference to a task held by the runtime.	STRONG The runtime holds a strong reference. *JavaScript* · *C♯* · *Tokio*	WEAK The runtime holds a weak reference. *Asyncio* · *Smol*	
Destruction How a task is cleaned up at the end of its extent.	AWAITED The task is awaited to completion. *JavaScript* · *Trio*	CANCELLED The task is cancelled, and then possibly awaited. *Swift* · *Tokio* · *Smol* · *Asyncio*	TERMINATED The program exits. *C♯*
Propagation What happens to exceptions in unawaited tasks.	DESTRUCTED Exceptions are reraised by dependents. *Trio*	NEVER The exception is kept within the task. *JavaScript* · *C♯* · *Tokio* · *Smol* · *Asyncio* · *Swift*	
Cancellation			
Awareness Whether a task is able to respond to being cancelled.	UNAWARE The task cannot respond to being cancelled. *Rust*	AWARE The task can respond to being cancelled. *Asyncio* · *Trio* · *Swift*	
Direction How cancellation is communicated through the task graph.	TOP-DOWN Starting from the root task, and communicated from dependents to dependencies. *Rust*	BOTTOM-UP Starting from the root's dependencies and communicated to dependents. *Asyncio* · *Trio*	SIMULTANEOUS To all transitive dependencies at once. *Swift*
Persistence [For AWARE cancellation] How long a cancellation of a task lasts.	TRANSIENT A task can ignore cancellation and proceed as normal. *Asyncio*	PERSISTENT A task can ignore cancellation but remains cancelled. *Trio* · *Swift*	

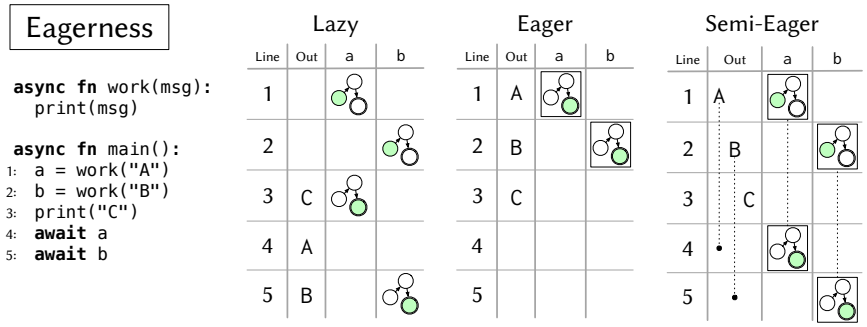


Fig. 4. An example program demonstrating the EAGERNESS dimension, executed with Rust (LAZY), C# (EAGER), and Swift (SEMI-EAGER) when using *async let*. Each column shows the line of execution, the text printed to stdout, and changes to the values of the variables a and b. Coroutines are represented as a state machine, with the active state colored in green. Tasks are represented as a boxed state machine. The dotted vertical lines represent nondeterminism, or an event that could happen anywhere within the indicated range.

An asynchronous function either implicitly (in the runtime implementation) or explicitly (in a user-visible way) desugars into an object that behaves like a coroutine. Upon entering the function, code runs until reaching an **`await`** point, when the function may yield to its caller (depending on its SUSPENSION, Section 3.1.2). Specifically, these objects behave like asymmetric stackless coroutines, to use the taxonomy provided by Moura and Ierusalimschy [31]. General coroutines can yield values on suspension and accept values on resumption. For straight-line asynchrony, a coroutine does not need to yield values on suspension, but needs only to accept values on resumption (or access them at a shared location).

When a coroutine representing an asynchronous computation is constructed via a function call, the EAGERNESS axis defines what happens next to the coroutine. In languages with first-class coroutines[1] such as Python [37] and Rust [50], we describe their behavior as LAZY: the coroutine is returned and no work occurs. In Figure 4, this means that `a = work("A")` constructs a coroutine in a starting state, but the coroutine is only executed at **`await`** `a` on line 4. In LAZY languages, to execute an async computation concurrently with its caller, one must use an explicit **`spawn`** operation to convert the coroutine into a *task* (e.g., as used in Figure 1).

A task is a handle on an async computation shared between its creator and an *async runtime*. The async runtime usually consists of a thread pool and some facilities for reacting to OS-level events by waking up waiting tasks. In languages without first-class coroutines, a call to an async function will return a task directly. However, languages differ with respect to scheduling of these tasks.

For EAGER languages such as C# [40, §15.14.3] and JavaScript [9, §27.7.5.1], on evaluating an async function application, the current thread transfers execution to the body of the async function. When reaching an await point, the function application evaluates to a task value, and control returns to the caller. In Figure 4, EAGER execution causes `work("A")` to behave like a normal function call, resulting in deterministic/single-threaded execution.

A language is SEMI-EAGER when an async function application immediately schedules the task onto a work queue, but allows the calling thread to continue executing past the function call. In Figure 4, SEMI-EAGER execution means that `work("A")` will run at some point after being called. The

[1]As of v1.92.0, Rust has first-class *futures* but not first-class coroutines. That is, calling an async function returns a value representing the computation, but that value does not offer a full coroutine interface. By contrast, Python has a stabilized coroutine interface implemented by its async functions. The Rust compiler does internally track async functions as coroutines, but this interface has not been stabilized.

task is only guaranteed to have finished by the end of `await a`, leading to nondeterministic/multi-threaded execution. We illustrate the SEMI-EAGER design using Swift [27] by assuming *async let* [21] bound async applications.[2] Unlike all other languages surveyed, a bare async application in Swift (e.g., `let a = work("A")`) results in a type error. Async applications must be immediately awaited and developers can use alternative syntaxes to run an async function within a new task. We classify Swift as being both EAGER and SEMI-EAGER as it provides alternative syntax for both semantics and doesn't itself have a default. For consistency, all pseudocode in this paper assumes that async function applications in Swift are bound using *async let*.

Design rationale. The choices in the EAGERNESS dimension offer a tradeoff in predictability, maximizing CPU multi-core utilization, and reducing task allocation. LAZY asynchrony has two main benefits. First, LAZY asynchrony can provide predictable memory allocation. Asynchronous function applications evaluate directly to a coroutine value that can be stack allocated. The developer must opt in to either boxing or spawning a coroutine. Predictable memory allocation is a key motivation for Rust's async/await design [51]. Second, by separating coroutines from tasks, the language can enable users to provide their own runtime. For example in Rust, this feature is useful so the runtime can be tailored to the needs of individual domains, such as embedded systems running Embassy [35] versus backend web servers running Tokio/Smol.

The downsides of the LAZY approach include: by offering an extensible runtime, async-related libraries in the ecosystem tend to have reduced interoperability. For example, in Rust, **`spawn`** is a runtime-specific primitive, so libraries using it must be coupled to a choice of runtime such as Tokio or Smol. Users must also deal with the complexity of understanding both coroutines and tasks. Users often find LAZY behavior surprising—to call a function and have nothing happen runs against a programmer's intuition transferred from synchronous function calls [7, 14, 26].

Non-LAZY approaches avoid these downsides by starting asynchronous work as soon as a function is called, either on the current thread (EAGER) or any thread (SEMI-EAGER). The EAGER approach reduces the overhead of control flow and context switches within the asynchrony runtime. The SEMI-EAGER approach leans more on the runtime, but in exchange provides a higher degree of concurrency by dispatching work to other threads. The increased concurrency can be particularly valuable for programs that want to avoid blocking a main thread, such as in UI applications [4, 20, 49].

3.1.2 SUSPENSION. Once asynchronous work has started, the next design divergence occurs at await points. The SUSPENSION dimension defines the guarantees a language makes about whether an await point is guaranteed to yield control. For example, consider the program in Figure 5 executing under an EAGER semantics. The line `a = spawn repeat("A")` will call into `repeat("A")`, which calls into `work("A")`, printing `"A"`, and returns to `await work("A")`. The task for `work("A")` has been completed, so a language has two choices: either yield control to the caller, or continue evaluation past the **`await`**. We call the former strategy STATIC SUSPENSION because an await point is statically guaranteed to always yield. We call the latter strategy DYNAMIC SUSPENSION because whether an await point yields depends on the awaited task.

In only one language surveyed, JavaScript, an await point is guaranteed to yield according to the ECMAScript specification [9, §27.7.5.3, step 3.b]. Therefore in Figure 5, running under JavaScript semantics, the program outputs "ABCAB" because **`await`** `work("A")` yields back to `main`. In all other languages, no such guarantee exists. For the example program operating under an EAGER

[2] Although we illustrate the SEMI-EAGER design using Swift, Swift is not SEMI-EAGER by default. To call an async function developers must choose between an immediately-awaited application (e.g., **`await`** `work("A")`), a SEMI-EAGER application (e.g., **`async`** `let x = work("A")`), or an EAGER application (e.g., `Task.immediate {` **`await`** `work("A") }`). A bare async application with no specification (e.g., `let x = work("A")`) is disallowed in Swift and results in a type error.

semantics such as in C♯, the program would output "AABBC" because **await** `work("A")` observes that `work("A")` is complete and therefore the await expression continues evaluation.

Design rationale. The two principal reasons to adopt STATIC SUSPENSION are preventing starvation and reducing non-determinism. In practice, some asynchronous function calls complete synchronously. For example, an asynchronous file read can return and buffer more bytes than requested; subsequent file reads within the buffered range return synchronously and perform no actual I/O. Under an EAGER application semantics with DYNAMIC SUSPENSION, a running task can starve the others by never suspending. Although a task's code contains await points, it can execute entirely synchronously, turning an I/O-bound task into a CPU-bound task. A STATIC SUSPENSION policy guarantees that tasks suspend at await points and don't accidentally starve others. Furthermore, STATIC SUSPENSION is more predictable because it eliminates the non-deterministic factor of asynchronous functions executing synchronously.

STATIC SUSPENSION comes at a performance cost. In the case of awaiting a completed task, STATIC SUSPENSION requires an expensive round-trip to the runtime to continue past the await. Languages with DYNAMIC SUSPENSION can mitigate the starvation issue by providing a `yield_now` primitive that has no effect but to force a suspension to the runtime.

Suspension

```
async fn main():
  a = spawn repeat("A")
  b = spawn repeat("B")
  print("C")
  await a
  await b
```

```
async fn work(msg):
  print(msg)

async fn repeat(msg):
  await work(msg)
  await work(msg)
```

Static

Execution tree	Out
a = spawn repeat("A")	
work("A")	
print("A")	A
await work("A")	
b = spawn repeat("B")	
work("B")	
print("B")	B
await work("B")	
print("C")	C
await a	
work("A")	
print("A")	A
await work("A")	
await b	
work("B")	
print("B")	B
await work("B")	

Dynamic

Execution tree	Out
a = spawn repeat("A")	
work("A")	
print("A")	A
await work("A")	
work("A")	
print("A")	A
await work("A")	
b = spawn repeat("B")	
work("B")	
print("B")	B
await work("B")	
work("B")	
print("B")	B
await work("B")	
print("C")	C
await a	
await b	

Fig. 5. An example program demonstrating SUSPENSION, executed with JavaScript (STATIC) and C♯ (DYNAMIC). Both languages are EAGER, but have different SUSPENSION semantics. Each column shows an execution trace and the order of prints to stdout; the traces are nested according to their level in the call stack.

3.2 End of Life

Much of the divergence between designs for straight-line asynchrony is about when and how tasks end. Recent designs often introduce some kind of "structured" asynchrony to ensure that tasks are accounted for at all times, and do not end up dangling or erroring unbeknownst to the rest of the program. This subsection covers the design dimensions of EXTENT (how long a task lives, Section 3.2.1), REFERENCE STRENGTH (the type of reference to tasks held by the runtime, Section 3.2.2), DESTRUCTION (what happens to destructed tasks, Section 3.2.3), and PROPAGATION (what happens to unawaited exceptions, Section 3.2.4).

3.2.1 EXTENT. Borrowing terminology from the Lisp community, the EXTENT of a task is the interval of time during which that task is capable of executing. For example, consider the program in Figure 6. What is the extent of the **spawn** `child_work()` task? The answer is it depends on quite a number of factors, as previously illustrated in Figure 1.

In some designs for straight-line asynchrony, especially earlier designs, tasks by default have an INDEFINITE EXTENT: the task can live until it is no longer reachable, at which point the task is

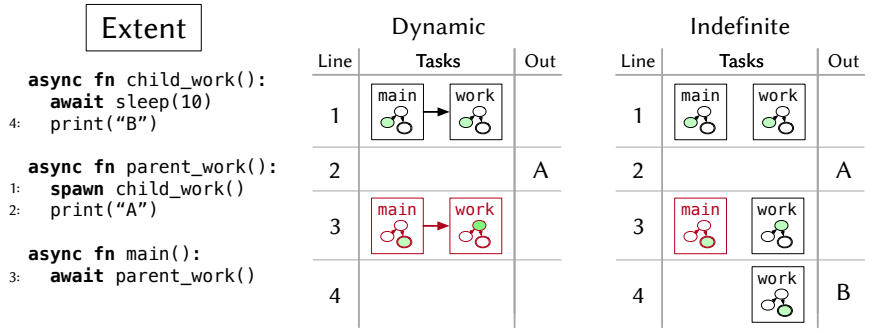


Fig. 6. An example program demonstrating the EXTENT dimension, executed with Swift (DYNAMIC) and JavaScript (INDEFINITE). Each column shows the task structure and output after executing each line. Arrows between tasks indicate a parent-child relationship used for DYNAMIC EXTENT.

destructed. JavaScript, C#, Python+Asyncio, and Rust+Tokio/Smol all use INDEFINITE EXTENT. Note that designs differ on how to determine reachability (see Section 3.2.2), and designs also differ on the mechanism for destruction (see Section 3.2.3). Figure 6 illustrates INDEFINITE EXTENT, showing how the `child_work()` task outlives its parent `parent_work()` task, and the program waits until the work is completed.

Some have called INDEFINITE EXTENT an "unstructured" approach to asynchrony [38], akin to using goto for control flow, as task handles can easily go unhandled, leaving asynchronous work dangling with no clear policy for managing it. This line of thought has led to the development of "structured" asynchrony approaches [10, 28, 43], which we describe as having DYNAMIC EXTENT, found in Swift and Python+Trio. With DYNAMIC EXTENT, a task's extent is tied to the extent of another task. In Figure 6, under DYNAMIC EXTENT, the `child_work()` task is associated with the `parent_work()` task. When the `parent_work()` task completes, its associated tasks are destructed, which means cancellation in the case of Swift. Therefore the second print never occurs.

Design rationale. The main argument for DYNAMIC EXTENT is that limiting tasks to known scopes leads to programs that are easier to reason about, and compose better with other language features [10, 28]. For instance in Python, DYNAMIC EXTENT can ensure that a task within a `with` block will not use resources (e.g., file descriptors) after they have been cleaned up [38]. However, existing implementations of DYNAMIC EXTENT only support a strictly hierarchical model of parent-child task relationships (via `TaskGroup` in Swift and `Nursery` in Trio). These implementations disallow programs that might have more complex dependencies between tasks, such as two parent tasks awaiting the same child task.

3.2.2 Reference Strength. For designs with INDEFINITE EXTENT, tasks live until no longer reachable. One lever available to these designs is to ask: does the runtime's reference to a task count? In the parlance of reference counting garbage collection, is the runtime's reference *strong* or *weak*? This choice constitutes the REFERENCE STRENGTH dimension.

For most INDEFINITE EXTENT designs, the runtime's handle is STRONG: JavaScript, C#, Rust+Tokio. Consequently, even if a task is no longer accessible from the main body of a program, it is still reachable by the runtime, and will therefore be executed to completion. Two designs use WEAK runtime handles: Rust+Smol and Python+Asyncio.

Rust lacks a garbage collector, so WEAK REFERENCE STRENGTH does not translate literally for Smol. In Smol, the task handle received from a **`spawn`** implements a destructor (via the `Drop` trait),

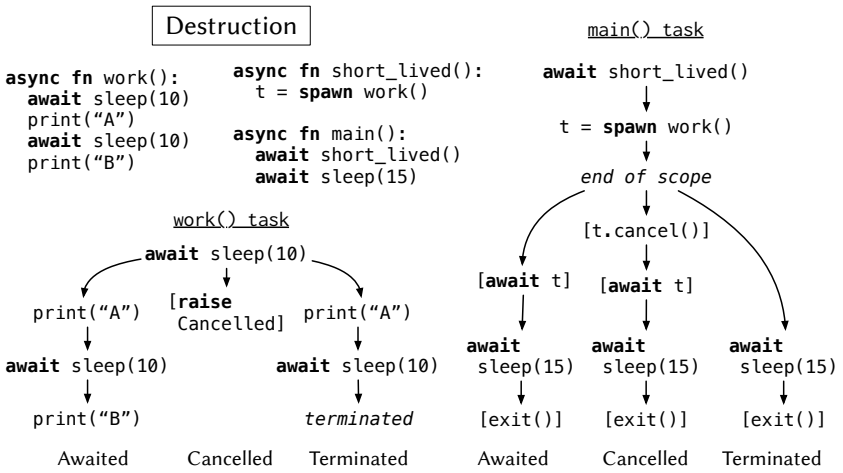


Fig. 7. An example program demonstrating the Destruction dimension. Each diagram shows the sequence of instruction execution for a task under Awaited (Python+Trio), Cancelled (Swift), and Terminated (C♯). The diagrams branch at the point of divergence between each approach. Administrative instructions that are executed but don't appear in the source code are displayed in brackets.

called when the handle goes out of scope (unless moved). The handle's destructor destructs the task, cancelling it in the case of Smol. In this way, Smol *effectively* achieves dynamic extent, unless the task is explicitly returned from the function. For example, the dynamic extent semantics for the program in Figure 6 are the same as Smol's semantics, and the indefinite semantics are the same as Tokio's semantics.

Asyncio also holds weak references to tasks. Unlike Smol's intentional design, Asyncio's weak reference approach is presently marked as a bug on the CPython repository, although the rationale for this design is unclear. According to Guido van Rossum [18]:

> I'm not sure why we designed it this way, and (from private email) @1st1 doesn't seem to recall either. But it was definitely designed with some purpose in mind – the code is quite complex and was updated every time an issue with it was discovered, and we've had many opportunities to change this behavior but instead chose to update the docs (python/cpython/pull/29163), even when asyncio itself suffered (python/cpython/issues/90467).

3.2.3 Destruction. Once a language has decided that it is time to destroy a task, we encounter yet another point of divergence: what does it mean to destroy a task? For example, consider the program in Figure 7. At the end of the extent of the spawned `work()` task (either end of `main()` for dynamic extent, or end of program for indefinite extent), what happens to the `work()` task? Today's languages offer three possibilities: to await the task, to cancel the task, or to terminate the task—the awaited, cancelled, and terminated points of the destruction dimension.

The destruction dimension is orthogonal to extent. JavaScript is a language with indefinite extent and awaited destruction—Figure 6 previously showed an example relying on this pair of design decisions. Python+Trio is a language/system with dynamic extent and awaited destruction, as tasks are tied to a "nursery," and the nursery waits until all its child tasks are completed before exiting. The awaited column in Figure 7 demonstrates Python+Trio semantics, where the program would print "ABC" by waiting for the `work()` task to finish before exiting `main()`.

Most other designs opt to cancel and then await[3] tasks at the end of EXTENT, such as Swift, Rust+Tokio, Rust+Smol, and Python+Asyncio. Figure 7 running under Swift would output nothing because `work()` is cancelled before it can reach a print statement. By contrast, C# will *terminate* tasks at the end of their INDEFINITE EXTENT, meaning the runtime simply exits while the destroyed tasks are running. In Figure 7, under C#, the program would print "A" because the task continues until being terminated after 15 seconds.

Design rationale. For languages with DYNAMIC EXTENT, the standard approach is to use CANCELLED DESTRUCTION. The idea is that responsible async code should await on a task in the same scope in which it was spawned. If a task is still executing after its parent scope exits, this is probably unintended, and it could create surprising behavior to wait until that task is completed. However, this approach makes less sense in Python+Trio, whose nurseries do not expose any concept of an awaitable task handle, so it is not trivial to directly await the result of a spawned task.

For languages with INDEFINITE EXTENT, any approach can make sense depending on the needs of the application, and it is straightforward to configure a single global runtime to adopt the desired behavior. For example, Rust+Tokio can be configured to terminate rather than await.

3.2.4 PROPAGATION. One final point of divergence at a task's end of life is the handling of exceptions. In all surveyed languages with exceptions (Python, Swift, JavaScript, C#), awaiting on an async function that has thrown an exception will re-raise the exception at the await point.[4] However, what happens to an exception raised in a task that is never awaited? We call this the PROPAGATION dimension.

Most languages NEVER propagate, meaning the exception generates no behavior outside its containing task, apart from perhaps printing to the console. Python+Asyncio and JavaScript will print a stack trace for an uncaught asynchronous exception. Node.js specifically will exit with a non-zero error code, although this latter behavior is not part of the ECMAScript specification so we still categorize it as a NEVER approach to PROPAGATION.

```
1  from trio import open_nursery
2
3  async def fail():
4    raise Exception()
5
6  async def main():
7    print("A")
8    async with open_nursery() as n:
9      n.start_soon(fail)
10   print("B")
```

Fig. 8. An example of DESTRUCTED PROPAGATION. Python+Trio reraises exceptions when their associated nursery scope ends.

Python+Trio is the only system which offers a different semantics, in part because tasks cannot be awaited, so the library must offer another mechanism for observing asynchronous exceptions. The example in Figure 8 (shown in Python, not pseudocode) will print "A" but not "B" because the nursery will re-raise the exception originating in the `fail` function. We describe this as DESTRUCTED PROPAGATION. A similar program in any other system will print "A" then "B", possibly with a stack trace printed in-between.

[3]The one exception here is Smol, which does not await cancelled tasks under its DYNAMIC EXTENT by default. The reason is that Smol implements task destruction using Rust's `Drop` trait, which only permits synchronous code, so Rust async implementations cannot await the completion of a task on drop. Smol offers an async `Task::cancel` method that implements the cancel+await semantics, but this method is not called automatically. Rust+Tokio, by contrast, does implement cancel+await because it uses INDEFINITE EXTENT, and it is easier to have a runtime await outstanding tasks at the end of the main function. Libraries like Python+Trio can implement await-on-destruction using Python's **`async with`** construct for scoped async entry/exit methods.

[4]Instead of exceptions, Rust uses the `Result` type to communicate errors. An `Err` variant is returned on await, which we consider a semantically equivalent design to re-raising an exception at the await point.

Design rationale. The upside to the DESTRUCTED approach is that an application is less likely to ignore problematic exceptions. With the NEVER approach, if a developer does not carefully monitor their logs, a task can fail and cause confusing bugs where expected operations never occur.

The downside is that the DESTRUCTED approach may run counter to either a developer's intuitions (i.e., exceptions aren't thrown away by the system) or their desired software architecture. Concurrency APIs often seek to encapsulate errors at thread or task boundaries, so errors in one thread cannot crash the whole system. For example, in a backend web server with many concurrent connections, it is likely undesirable that one exceptional connection should crash the entire server. Either way, it seems sensible that a straight-line asynchrony API should offer *some* way of accessing information about unawaited exceptions, rather than only printing to standard error, or ignoring unawaited exceptions outright.

3.3 Cancellation

As demonstrated by the case study on `timeout` (Section 2.3), cancellation is both a key component and a complicating factor in straight-line asynchrony. A naive cancellation approach would abruptly kill a running task, but no language adopts this approach because the task may be holding a mutex, writing to a file, or otherwise in a critical region with respect to a system's invariants.

A straightforward approach to cancellation in user-space is to use a *cancellation token*, as demonstrated in Figure 3a. The developer is responsible for threading the token through all the relevant async code, and they are also responsible for reacting to the token in any async code that should be cancellable. Cancellation token APIs can be found in JavaScript, C♯, and Rust. However, the amount of developer effort required for this approach is prohibitive. Modern takes on straight-line asynchrony have therefore sought to develop alternative cancellation approaches to lower the developer burden.

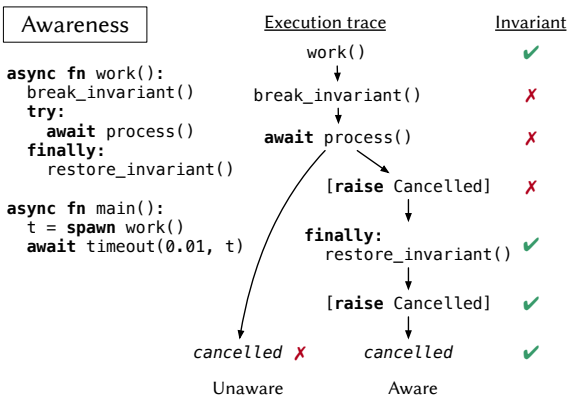


Fig. 9. An example program demonstrating the AWARENESS dimension. A branching execution trace is paired with an indicator of whether an invariant is satisfied. An AWARE approach allows the cancelled task to restore the invariant before proceeding with cancellation, while an UNAWARE approach remains cancelled with the invariant broken.

This subsection describes the two principal design dimensions of non-token-based, or *integrated*, cancellation approaches, AWARENESS (whether a task can react to cancellation, Section 3.3.1) and DIRECTION (how cancellation propagates through related tasks, Section 3.3.2). This subsection focuses on languages with integrated cancellation designs, namely Rust, Swift, and Python.

3.3.1 Awareness. Integrated cancellation approaches all center around triggering cancellations at await points. A key distinction is whether a cancelled task is aware that it is being cancelled, in order to handle or ignore the cancellation at a given await point.

The most straightforward approach is UNAWARE cancellation, where a task has no opportunity to react to being cancelled. This approach is taken by Rust in both Tokio and Smol. In Rust, cancellation essentially means never executing a coroutine again after it has yielded. With a direct handle on the coroutine, this simply means not polling the coroutine, or explicitly dropping it. If the coroutine

is within a task, then the task must be marked as cancelled, and the runtime will similarly refuse to reschedule the task.

By contrast, Python and Swift both implement a form of AWARE cancellation. In Python (both Asyncio and Trio), when a task is cancelled, a cancellation exception is thrown into the task. An async function can wrap any await point in a try/except block which can catch this cancellation exception, performing cleanup or consuming the exception. In Swift, when a task is cancelled, a flag is set indicating that the task is cancelled. Functions can then opt in to raising a cancellation exception when this flag is set. It is encouraged to indicate a cancelled task by raising a `CancellationError`, which the standard library `Task.sleep` does.

Design rationale. A benefit of UNAWARE cancellation is that tasks can be reliably cancelled. The principal case that delays UNAWARE cancellation is a compute-bound task which does not cooperatively yield to the runtime. With AWARE cancellation, functions can either ignore cancellation or fail to opt in, which we discuss further in Section 3.3.2.

A major drawback to UNAWARE cancellation is that cancellation can easily violate logical invariants of a program. For example, consider the program in Figure 9. Say a program wants to do asynchronous work in a critical region where an invariant is broken and later restored. With UNAWARE cancellation, the `work` task can be halted at **`await`** `process()`, causing the invariant to remain broken. In Rust, this is a widely-documented problem with its cancellation system[5] [33, 34]. With AWARE cancellation through exceptions, the `finally` block has the opportunity to restore the invariant before continuing to raise the cancellation exception.

3.3.2 DIRECTION. Tasks form a dependency graph of dependents, tasks awaiting on others, and dependencies, tasks being awaited. Cancellation starts at a specific node, raising the question: how does cancellation proceed through the graph? We call approaches to this question the DIRECTION dimension.

One approach is TOP-DOWN cancellation, where cancellation is communicated from dependents to dependencies. This approach describes Rust, which leans on the `Drop` trait to implement cascading communication during cancellation. In Smol, when a task is cancelled and later dropped, that drops the handles to all of that task's children, which in turn cancels those tasks, and so on through the task graph. This cancellation approach is feasible in Rust because it has a deterministic drop order, i.e., nodes have a well-defined parent / child relationship, where parents have an owning reference to the child.

Another approach is BOTTOM-UP cancellation, where cancellation is communicated from dependencies to dependents. This approach describes Python, which leans on exceptions to communicate cancellations. When a task is cancelled, both Asyncio and Trio will walk the task graph to find its leaf nodes, and raise a cancellation exception at those tasks. That exception is then raised across await points to async callers, which can choose to either catch the exception or allow it to propagate.

The final approach is SIMULTANEOUS cancellation, where cancellation is communicated at once to all transitive dependencies of the cancelled task. This approach describes Swift, which uses a shared cancellation flag. In Swift, convention holds that async functions should observe the cancellation flag and then raise a cancellation exception. A key difference from BOTTOM-UP cancellation is that the flag ensures that a task is aware of being cancelled, even if the cancellation exception was caught and consumed.

[5] It is possible in Rust to implement something similar to the `finally` block using the `Drop` trait. When a task is cancelled, the task is eventually dropped and drop handlers are invoked for all coroutines in the task. One can theoretically define a custom `Drop` implementation for a bespoke type that calls `restore_invariant`. However, this approach is verbose (requiring a newtype and impl), unwieldy (requires mutable references to data structures, complicating ownership), and synchronous-only (one cannot call async code in a `Drop` implementation without blocking).

Design rationale. The TOP-DOWN approach is a straightforward means of propagating cancellation without needing a mechanism like exceptions, which is especially useful for exception-less languages such as Rust. As previously mentioned, a TOP-DOWN approach requires the dependency graph to have a deterministic top-down order. The main limitation of the TOP-DOWN approach is that it makes custom handling of cancellations—such as running cleanup code—difficult: a cancelled task is never resumed, so it has no opportunity to observe its own cancellation, and cleanup is limited to synchronous destructors (e.g., Rust's `Drop` trait).

The BOTTOM-UP and SIMULTANEOUS approaches both provide the opportunity for handling cancellations. The major difference is that the BOTTOM-UP approach provides individual async functions more leeway to ignore cancellation, irrespective of the calling context. With the SIMULTANEOUS approach, every task in a task tree is guaranteed to know if it is cancelled. As implemented in Swift, this approach requires more complexity—cancellation is represented both formally through the cancellation flag, and conventionally through a cancellation exception observed by async primitives. In exchange, async functions can avoid the possibility of missed cancellations.

3.3.3 Persistence. One final element of cancellation is its PERSISTENCE: when a task is cancelled, does the cancellation request persist (e.g., subsequent async work is also marked as cancelled), or is cancellation requested once? We will describe these options as PERSISTENT cancellation and TRANSIENT cancellation. The PERSISTENCE dimension is important for cancel AWARE languages, because on cancellation a task may perform additional async work.

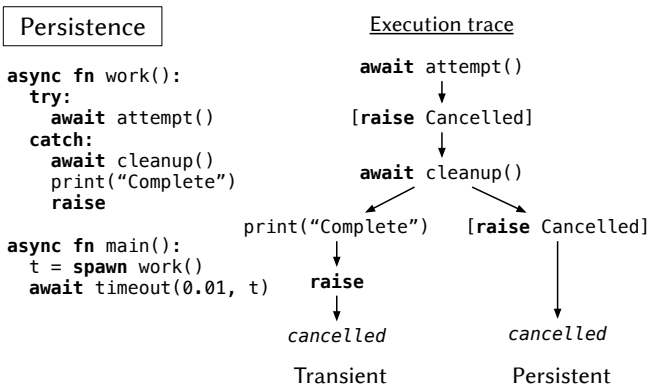


Fig. 10. An example program demonstrating PERSISTENCE. After the `work` task is cancelled, then the first await raises a cancellation exception. Under TRANSIENT semantics (Python+Asyncio), subsequent awaits proceed as normal. Under PERSISTENT semantics (Python+Trio), subsequent awaits still raise a cancelled exception, stopping further execution.

Swift and Python+Trio use PERSISTENT cancellation. Swift and Trio both implement a cancellation flag shared amongst a task graph, and that flag is permanently active once set. Swift's flag is part of its public API, while Trio's flag is an implementation detail. Consequently, if an async computation ignores its cancellation, then subsequent awaits may again raise a cancellation exception; for both Swift and Trio any subsequent await point may result in a cancellation exception.

Rust and Python+Asyncio use TRANSIENT cancellation. In Rust's case, because cancellation is UNAWARE, cancellation is unrecoverable. For Asyncio, cancellation is recoverable, so if an async computation ignores a cancellation request, then it can continue spawning and awaiting tasks unimpeded. Importantly, tasks spawned after the initial cancellation request would be unaware of the previous cancellation attempt.

Design rationale. For UNAWARE cancellation, PERSISTENCE is a moot point, so the main question is which design best fits AWARE cancellation. This perhaps comes down to a philosophy on why tasks would choose to ignore their cancellation, and what risks are incurred as a result.

$$\begin{aligned}
&\text{Number } n \quad \text{String } s \quad \text{Variable } x \\
&\text{Const } c ::= \textbf{true} \mid \textbf{false} \mid n \mid s \\
&\text{Expr } e ::= x \mid v \mid e\ e \mid \textbf{reset}\ e \\
&\qquad \mid \textbf{shift}\ x \to e \mid \cdots \\
&\text{Value } v ::= c \mid () \mid \textbf{fn}\ x \to e \mid \cdots \\
&\text{Eval Ctx } E ::= [\cdot] \mid v\ E\ e \mid \textbf{reset}\ E \mid \cdots \\
&\text{Heap } \sigma ::= \cdot \mid \sigma, x \mapsto v
\end{aligned}$$

(a) A subset of the λ_v grammar.

$$e ::= \ldots \mid \textbf{throw-in}\ e\ e \mid \textbf{throw}\ e \mid \textbf{try}\ e\ \textbf{catch}\ e$$

(b) The extension grammar for λ_{exn}, which extends λ_v with exception handling forms, including **throw-in**, which we model after CPython's `.throw()` coroutine method.

$$\langle \sigma, E[\textbf{throw-in}(\textbf{fn}\ x \to E_{inner}[x], v)] \rangle \quad \text{[Throw-In]}$$
$$\longrightarrow \langle \sigma, E[E_{inner}[\textbf{throw}\ v]] \rangle$$

(c) An example reduction rule for the λ_{exn} extension language.

Fig. 11. An illustrative subset of the grammar and reduction rules for the core calculi λ_v and λ_{exn}.

The main argument for TRANSIENT cancellation is for tasks to perform async resource cleanup. For example, as shown in Figure 10, if `work()` is cancelled in the middle of `attempt()`, then it can still call **`await`** `cleanup()` before re-raising the cancellation exception. A concrete instance of this pattern is that closing a TLS socket may require asynchronously sending a `close_notify` alert to the peer so that they can be cryptographically assured that you intended to close the socket (docs.openssl.org). However, TRANSIENT cancellation has a greater risk for cancellation signals to go unnoticed. If one task suppresses a cancellation signal, then all others may never become aware of the attempt.

Runtimes with PERSISTENT cancellation typically come with a cancellation *shielding* operator to allow for asynchronous cleanup. For example, the Python+Trio primitive `move_on_after` (demonstrated in Figure 2b) exposes a `shield` keyword argument that, when `True`, suppresses cancellation signals in nested code. This allows async cleanup code to temporarily ignore cancellation while still ensuring liveness by moving on after a fixed duration.

4 Formal Model

The preceding taxonomy helps us make sense of the space of straight-line asynchrony, but it does not provide an account of each language's async holistically. In this section we model the core async features of each of the languages under study. An example of the payoff is Figure 17, which precisely accounts for the divergent behaviors of Figure 1. The rest of the section explains the key components of the figure.

Rather than describe each language separately, our semantics is designed to both capture the commonalities and highlight the differences between async in these languages. To do this, we unify the languages' implementation of asynchrony using delimited continuations rather than faithfully modeling each implementation using state machines and other such mechanisms. The full semantics is too large to include in its entirety; we provide the elided rules in the Supplemental Material.

We provide an executable version of the model in Redex [11], including a full test suite that covers all the examples from Sections 2 and 3 for all runtimes. The model includes a fuzzer that generates random programs, compiles them into the respective languages, and checks that the output from the real program and model program match. Due to non-determinism, we run each program $N = 50$ times and check that the program outputs are a subset of model outputs. This gives us high confidence that our model matches the behavior of the corresponding runtimes.

4.1 Base Language

Figures 11a and 11b provide the grammar and semantics of the core calculi that comprise our base for the async extensions. The core language λ_v contains mutable references and delimited

$$
\begin{array}{rcl@{\qquad}rcl}
\text{Time } t & \in & \mathbb{N}_0 \cup \{\infty\} & \text{Frame } F & ::= & \langle \ell, e \rangle \\
\text{Label } \ell & ::= & x \mid \textbf{sync} & \text{Frame Stack } FS & ::= & \epsilon \mid F \circ FS \\
\text{Continuation } \kappa & ::= & \textbf{fn } x_{unit} \longrightarrow e & \text{Process } P & ::= & \{FS_1, \ldots, FS_n\} \\
\text{Signal Queue } T & ::= & \epsilon \mid (t, \ell, \kappa) \circ T & \text{Task Queue } Q & ::= & \epsilon \mid (\ell, \kappa) \circ Q
\end{array}
$$

$$
e \quad ::= \quad \cdots \mid \textbf{sys-io}(e, e) \mid \textbf{sys-start-soon}(\ell, e) \mid \textbf{sys-start-later}(e, \ell, e)
$$

(a) An illustrative subset of the λ_{async} extension grammar.

C♯ / JavaScript / Rust / Python

$e ::= \ldots \mid \textbf{await } e$

$v ::= \ldots \mid \textbf{async fn } x \longrightarrow e$

Asyncio / Trio / Smol / Tokio

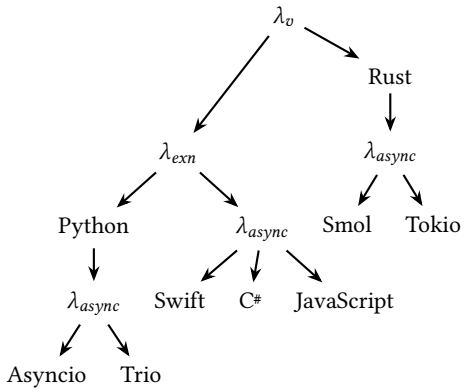
$e ::= \ldots \mid \textbf{spawn } e \mid \textbf{cancel } e$

Swift

$e ::= \ldots \mid \textbf{await } e \mid \textbf{cancel } e \mid \textbf{cancelled?}$

$v ::= \ldots \mid \textbf{async fn } x \longrightarrow e$

(b) The grammar extensions for each language.



(c) The order of extension for each async runtime.

Fig. 12. The grammar extensions and order of extension for λ_{async} and the seven async systems under study. Python and Rust are *extended by* λ_{async} because they provide native coroutines. The runtimes for those languages are defined as libraries. The other languages: Swift, JavaScript, C♯, *extend* λ_{async} because they do not provide native coroutines and their async runtimes are built into the language runtime.

continuations via the **shift** and **reset** operators described by Danvy and Filinski [8]. Figure 11a provides the grammar for λ_v. Reduction rules for synchronous grammars are defined over a mutable heap, σ, and an expression e, $\langle \sigma, e \rangle$.

For languages with exceptions, we extend λ_v with **try**/**catch** and **throw** forms and refer to this language as λ_{exn}. Figure 11b provides the grammar extension for λ_{exn}, and Figure 11c provides the reduction rule for the nonstandard form **throw-in**, which we model after Python's `.throw()` coroutine method. The form **throw-in** resumes a continuation with an exception instead of a value.

4.2 The Async Platform

We now describe the configuration of the shared async grammar, which can extend any λ_v-based language. Figure 12a defines the shared grammar constructs that encode the async runtime building blocks discussed in Section 3. We will refer to this extension as the async platform, λ_{async}.

We introduce the notion of an async *frame*, F, written $\langle \ell,\ e \rangle$, which consists of an expression e, along with a frame label, ℓ.[6] A *frame label*, ℓ, is either the name of the task this frame corresponds to, or *sync* to denote a synchronous frame. A *frame stack*, FS, is a list of frames—essentially a thread. An empty frame stack is written ϵ, and we write $F \circ FS$ to denote a frame stack whose head is F and tail is FS. A *process*, P, is a collection of frame stacks. We will use the syntax $P \uplus FS$ to select one frame stack in P to run. A *task queue*, Q, stores the async work waiting to be run; Q comprises tuples that have a label and a continuation, (ℓ, κ). A *signals queue*, T, stores async work that is

waiting on I/O operations. T comprises tuples that have a time, a label, and a continuation, (t, ℓ, κ), κ, is invoked no earlier than timestamp t. We will use the same stack notation for Q and T as we do for FS. We will additionally write $Q \triangleleft (\ell, \kappa)$ to append to the end of the queue.

The async platform provides a few additional administrative instructions that we will describe here. The **sys-block**(e) expression provides the barrier between synchronous code and asynchronous code. Languages may use a block expression to block a synchronous frame until the asynchronous expression has completed running. The block expression marks the runtime entry (e.g., creating a thread pool), and when it returns, the runtime exit (e.g., destroying the thread pool and runtime queues).

The expression **sys-io**(e_t, e) is the I/O operation provided by the runtime system. It models real-world I/O operations that would take some non-deterministic amount of time to resolve, like receiving packets from the network or querying a database. The form is prescient because the result is provided upfront: e.g., **sys-io**$(5, \text{“hello”})$ creates a task that will resolve to the value “hello” after at least five time steps. Furthermore, we model I/O operations that may not complete as **sys-io**(∞, e). The **sys-io** form is deterministic in its evaluation *value*, but not in its evaluation *time*.

The expression **sys-start-soon**(ℓ, κ) appends the continuation κ to Q, set to run in a frame with the specified label ℓ. The expression **sys-start-later**(ℓ, κ) appends the continuation to the signal queue, T, which will be scheduled after timestep t.

The grammar extensions for the remaining languages are shown in Figure 12b, and the order in which we layer the extensions in Figure 12c. The reduction rules for all async-capable languages, which base Python and Rust are not, are defined over the tuple $\langle t, \sigma, Q, T, P \rangle$. Most tuple items remain constant in many rules; to simplify the presentation we will use a double angle notation $\langle\!\langle \ldots \rangle\!\rangle$ to indicate that unmodified tuple items are elided.

The goal of the operational semantics is to highlight the differences between languages. We will therefore present the same reduction rule from different languages together. We will also call attention to the portion relevant to each design dimension with a highlight in the following colors: EAGERNESS, EXTENT, DESTRUCTION, PROPAGATION, and SUSPENSION, and within cancellation: AWARENESS, DIRECTION, and PERSISTENCE. For space reasons we do not present the rules relevant for REFERENCE STRENGTH, and we simplify the presentation with a liberal use of metafunctions and macros, see Table 2 for a description of those used; the full definitions of grammars, reductions, and metafunctions are provided in the Supplemental Material.

4.3 Start/End of Life

The rule [Async-App] defines how a coroutine/task is constructed and, for non-LAZY languages, what a task does at the end of its computation. We will use the definitions in Figure 13 to highlight the semantics for the start and end of a computation's life.

Figure 13 provides the [Async-App] rule for EAGER, SEMI-EAGER, and LAZY async applications. For EAGER and SEMI-EAGER languages an async application follows a similar procedure: a new task, x_{task}, is allocated in the global store; evaluation of the function body is wrapped in a try/catch block, on successful evaluation the result is stored in the task and it's marked as completed, on exception the error value is stored in the task and it's marked as failed. After the task is marked as completed or failed, all dependent tasks, those awaiting on the current task, are scheduled to resume executing with the **sys-start-soon** form. EAGER and SEMI-EAGER languages differ by where the new e_{task} expression is placed. EAGER languages set the expression in a new frame on *top* of the current frame stack. SEMI-EAGER languages place the expression in the task queue, Q, to run soon

[6]We use a labeled async frame to explicitly associate executing code with a task. An alternative formalism would be to delimit tasks using a labeled context. We view these two options as equivalent and chose the former.

Table 2. Metafunctions and macros used in the reduction rules. Metafunctions evaluate to a term, while macros expand into expressions. We set macros in italics to distinguish them from metafunctions.

Metafunction	Description
task-cancelled : $\sigma \times \ell \to b$	*true* iff task ℓ, or any transitive dependent, is cancelled
task-alloc : $\sigma \to (\sigma, x)$	Allocate a new task struct bound under gensymed name x in σ
task-uncancel : $\sigma \times \ell \to \sigma$	Remove cancellation from task ℓ in σ
task-add-dependency : $\sigma \times \ell_1 \times \ell_2 \to \sigma$	Add task ℓ_2 as a dependency of ℓ_1
Macro	**Description**
task-add-dependent : $v \times \ell \times e \rightsquigarrow e$	Register ℓ (continuation e) as a dependent of awaitable v
task-set-done : $x \times e \rightsquigarrow e$	Evaluate e and use value as x's resolved success value
task-set-failed : $x \times e \rightsquigarrow e$	Evaluate e and use value as x's resolved failure value
task-is-completed : $e \rightsquigarrow e$	*true* iff task is resolved (success or failure)
task-get-result : $e \rightsquigarrow e$	Return task's final value; reraise if task failed
task-continue-with : $v \times x \rightsquigarrow e$	Resume continuation bound by x with settled value of task v
task-get-dependents : $e \rightsquigarrow e$	Return dependents of the task that e evaluates to
task-cancel-dependencies : $x \rightsquigarrow e$	Cancel task x and its descendants
task-wait-on-dependencies : $x \rightsquigarrow e$	Wait for all dependencies of x to settle
task-reraise-dependency-failures : $x \rightsquigarrow e$	Reraise dependency failures

by the next available thread. The semantics for the LAZY languages diverge significantly. Neither Rust nor Python has a built-in async runtime, so their semantics are not defined as an extension of λ_{async}. The LAZY [Async-App] variant simply captures the function body as a continuation. We will discuss the [Spawn] rule—used to turn a coroutine into a task—below.

[Async-App] for the (semi-)eager languages also defines the end of life for a task, including the EXTENT, DESTRUCTION, and PROPAGATION semantics. When C#/JavaScript allocate a task, it is independent of the current context, meaning that the new task has no relationship to the currently executing task (labelled ℓ). The absence of a relationship between the two tasks marks the INDEFINITE EXTENT semantics for these two languages. In contrast, allocated tasks in Swift are added as dependencies of the currently executing task. Furthermore, within a Swift task body, e_{task}, after the task is completed or failed its dependencies are cancelled and waited on. The dependency between the two tasks and the added operations at the end of the task body marks the different semantics in Swift along the EXTENT and DESTRUCTION dimensions. Although Swift cancels and waits for dependencies to finish, exceptions held by *failed* tasks are not propagated.

The rule [Spawn] is provided by the runtimes of LAZY languages to elevate a coroutine to a task registered in the runtime. [Spawn] defines the semantics for the task start and end of life that are absent from the LAZY definition of [Async-App].

[Spawn] is defined separately for languages with INDEFINITE and DYNAMIC EXTENT. The definition used by Asyncio/Tokio/Smol allows for INDEFINITE EXTENT. The rule follows the semantics discussed previously for C#'s [Async-App] rule. Trio enforces DYNAMIC EXTENT and the rule follows closely that discussed previously for Swift. However, after the task body, e_{task}, finishes executing, dependencies are waited on *uncancelled* to finish, and exceptions from failed dependencies are reraised. These differences mark Trio's unique combination of AWAITED DESTRUCTION and DESTRUCTED PROPAGATION semantics.

Figure 15 provides the definitions for the rule [Await]. JavaScript, the only language sampled with STATIC SUSPENSION semantics, always captures the current delimited continuation and adds itself as a dependent on the awaited value. The captured continuation, e_k, is returned. Languages

$$\langle\!\langle \sigma_0, P \uplus \{ \langle \ell, E[(\textbf{async fn}\ x \to e_{body})\ v]\rangle \circ FS\}\rangle\!\rangle \quad \text{[C♯/JavaScript-Async-App]}$$
$$\longrightarrow \langle\!\langle \sigma_2, P \uplus \{\langle x_{task}, e_{task}\rangle \circ \langle \ell, E[x_{task}]\rangle \circ FS\}\rangle\!\rangle$$
$$\text{where } x_{fresh} = \text{gensym}(),\ \sigma_1 = \sigma_0[x_{fresh} \mapsto v],\ e_{subst} = e_{body}[x_{fresh}/x]$$
$$(\sigma_2, x_{task}) = \text{task-alloc}(\sigma_1)$$
$$e_{task} = \textbf{reset}\ (\ \textbf{try}\ \mathit{task\text{-}set\text{-}done}(x_{task}, e_{subst})\ \textbf{catch}\ v_{err} \to \mathit{task\text{-}set\text{-}failed}(x_{task}, v_{err})$$
$$\textbf{sys-start-soon}(\mathit{task\text{-}get\text{-}dependents}(x_{task})))$$

$$\langle\!\langle \sigma_0, Q, P \uplus \{\langle \ell, E[(\textbf{async fn}\ x \to e_{body})\ v]\rangle \circ FS\}\rangle\!\rangle \quad \text{[Swift-Async-App]}$$
$$\longrightarrow \langle\!\langle \sigma_3, Q \lhd (x_{task}, \kappa), P \uplus \{\langle \ell, E[x_{task}]\rangle \circ FS\}\rangle\!\rangle$$
$$\text{where } x_{fresh} = \text{gensym}(),\ \sigma_1 = \sigma_0[x_{fresh} \mapsto v],\ e_{subst} = e_{body}[x_{fresh}/x]$$
$$(\sigma_2, x_{task}) = \text{task-alloc}(\sigma_1)$$
$$\sigma_3 = \text{task-add-dependency}(\sigma_2, \ell, x_{task})$$
$$e_{task} = \textbf{reset}\ (\ \textbf{try}\ \mathit{task\text{-}set\text{-}done}(x_{task}, e_{subst})\ \textbf{catch}\ v_{err} \to \mathit{task\text{-}set\text{-}failed}(x_{task}, v_{err});$$
$$\mathit{task\text{-}cancel\text{-}dependencies}(x_{task});$$
$$\mathit{task\text{-}wait\text{-}on\text{-}dependencies}(x_{task});$$
$$\textbf{sys-start-soon}(\mathit{task\text{-}get\text{-}dependents}(x_{task})))$$
$$\kappa = \textbf{fn}\ x_{unit} \to x_{unit}\ ;\ e_{task}$$

$$\langle \sigma_0, E[(\textbf{async fn}\ x \to e_{body})\ v]\rangle \quad \text{[Rust/Python-Async-App]}$$
$$\longrightarrow \langle \sigma_1, E[\textbf{reset}\ (\textbf{shift}\ k \to k\ ;\ e_{subst})]\rangle$$
$$\text{where } x_{fresh} = \text{gensym}(),\ \sigma_1 = \sigma_0[x_{fresh} \mapsto v],\ e_{subst} = e_{body}[x_{fresh}/x]$$

EAGERNESS EXTENT DESTRUCTION PROPAGATION

Fig. 13. A comparison of the rules [Async-App] across languages.

with DYNAMIC SUSPENSION first check if the awaited value is ready—if it is, no suspension occurs. If a value is not ready, the continuation is captured and registered as a dependent of the pending computation; e_k is returned.

4.4 Cancellation

Figure 16 provides the full set of rules for the cancellation mechanism provided by Swift, Tokio, Smol, Asyncio, and Trio. No rules are present for C♯ or JavaScript as they do not provide a built-in mechanism to cancel a running task (see Section 3.3). The design dimensions for cancellation are AWARENESS, DIRECTION, and PERSISTENCE.

The [Cancelled?] rule is provided for Swift to allow a running task to check if it has been cancelled. This form defines both the AWARE AWARENESS and SIMULTANEOUS DIRECTION that is unique to Swift. A task does not need to rely on another mechanism to signal cancellation; this information is communicated between the runtime directly to each task.

The [Schedule-Cancelled] rules for Asyncio/Trio are almost identical. When a cancelled task is rescheduled the runtime will resume the continuation with an exception, using our **throw-in** form. By resuming the computation they are AWARE of their cancellation. The cancellation DIRECTION flows BOTTOM-UP because the tasks in the task queue, Q, are depended on by others and not awaiting for any dependencies; in other words, they are the deepest dependencies.

Where Asyncio and Trio diverge is cancellation PERSISTENCE. A peculiarity of Asyncio is the TRANSIENT PERSISTENCE of cancellation: after a cancelled task is resumed with a cancellation signal,

$$\langle\!\langle \sigma_0, Q, \quad P \uplus \{\langle \ell, E[\textbf{spawn } v_{coro}]\rangle \circ FS\}\rangle\!\rangle \qquad \text{[Asyncio/Tokio/Smol-Spawn]}$$
$$\longrightarrow \langle\!\langle \sigma_1, Q \lhd (x_{task}, \kappa), P \uplus \{\langle \ell, E[x_{task}]\rangle \circ FS\}\rangle\!\rangle$$
$$\text{where } (\sigma_1, x_{task}) = \text{task-alloc}(\sigma_0)$$
$$e_{task} = \textbf{reset } (\textbf{try } \mathit{task\text{-}set\text{-}done}(x_{task}, \textbf{await } v_{coro}) \textbf{ catch } v_{err} \to \mathit{task\text{-}set\text{-}failed}(x_{task}, v_{err}); \textbf{sys-start-soon}(\mathit{task\text{-}get\text{-}dependents}(x_{task})))$$
$$\kappa = \textbf{fn } x_{unit} \to x_{unit}\ ;\ e_{task}$$

$$\langle\!\langle \sigma_0, Q, \quad P \uplus \{\langle \ell, E[\textbf{spawn } v_{coro}]\rangle \circ FS\}\rangle\!\rangle \qquad \text{[Trio-Spawn]}$$
$$\longrightarrow \langle\!\langle \sigma_2, Q \lhd (x_{task}, \kappa), P \uplus \{\langle \ell, E[x_{task}]\rangle \circ FS\}\rangle\!\rangle$$
$$\text{where } (\sigma_1, x_{task}) = \text{task-alloc}(\sigma_0)$$
$$\sigma_2 = \text{task-add-dependency}(\sigma_1, \ell, x_{task})$$
$$e_{task} = \textbf{reset } (\textbf{try } \mathit{task\text{-}set\text{-}done}(x_{task}, \textbf{await } v_{coro}) \textbf{ catch } v_{err} \to \mathit{task\text{-}set\text{-}failed}(x_{task}, v_{err}); \mathit{task\text{-}wait\text{-}on\text{-}dependencies}(x_{task}); \mathit{task\text{-}reraise\text{-}dependency\text{-}failures}(x_{task}); \textbf{sys-start-soon}(\mathit{task\text{-}get\text{-}dependents}(x_{task})))$$
$$\kappa = \textbf{fn } x_{unit} \to x_{unit}\ ;\ e_{task}$$

EAGERNESS EXTENT DESTRUCTION PROPAGATION

Fig. 14. A comparison of [Spawn] across Rust/Python runtimes. The **try/catch** is colored gray because it is present in the Asyncio semantics, but not those for Tokio/Smol—the rules are otherwise equivalent.

$$\langle\!\langle P \uplus \{\langle \ell, E[\textbf{await } v_{aw}]\rangle \circ FS\}\rangle\!\rangle \qquad \text{[C\#/Swift/Asyncio/Trio/Smol-Await]}$$
$$\longrightarrow \langle\!\langle P \uplus \{\langle \ell, E[e_{await}]\rangle \circ FS\}\rangle\!\rangle$$
$$\text{where } e_{await} = \textbf{if } \mathit{task\text{-}is\text{-}completed}(v_{aw}) \textbf{ then } \mathit{task\text{-}get\text{-}result}(v_{aw}) \textbf{ else shift } k \to \mathit{task\text{-}add\text{-}dependent}(v_{aw}, \ell, \mathit{task\text{-}continue\text{-}with}(v_{aw}, k))$$

$$\langle\!\langle P \uplus \{\langle \ell, E[\textbf{await } v_{aw}]\rangle \circ FS\}\rangle\!\rangle \qquad \text{[JavaScript-Await]}$$
$$\longrightarrow \langle\!\langle P \uplus \{\langle \ell, E[e_{await}]\rangle \circ FS\}\rangle\!\rangle$$
$$\text{where } e_{await} = \textbf{shift } k \to \textbf{if } \mathit{task\text{-}is\text{-}completed}(v_{aw}) \textbf{ then } \mathit{task\text{-}get\text{-}result}(v_{aw}) \textbf{ else } \mathit{task\text{-}add\text{-}dependent}(v_{aw}, \ell, \mathit{task\text{-}continue\text{-}with}(v_{aw}, k))$$

SUSPENSION

Fig. 15. A comparison of [Await] semantics that define DYNAMIC and STATIC SUSPENSION. DYNAMIC suspending languages only suspend when the awaited value isn't immediately ready.

it is no longer marked as cancelled. Further async work can be done and a second cancellation signal will not be raised. Trio and Swift both have PERSISTENT cancellation, so further async work in an exception handler would be resumed with another cancellation exception.[7]

The Rust runtimes, Tokio and Smol, do not provide cancellation AWARENESS. Cancelled tasks are not rescheduled but deallocated: the continuation κ is thrown away and the task is set as failed by the expression e_{fail}.

$$\langle\!\langle P \uplus \{\langle \ell, E[\textbf{cancelled?}()]\rangle \circ FS\}\rangle\!\rangle \qquad \text{[Swift-Cancelled?]}$$
$$\longrightarrow \langle\!\langle P \uplus \{\langle \ell, E[e_{bool}]\rangle \circ FS\}\rangle\!\rangle$$
$$\text{where } e_{bool} = \text{task-cancelled}(\sigma, \ell)$$

$$\langle\!\langle (\ell, \kappa) \circ Q, P \uplus \{\epsilon\}\rangle\!\rangle \qquad \text{[Trio-Schedule-Cancelled]}$$
$$\longrightarrow \langle\!\langle Q, P \uplus \{\langle \ell, \textbf{throw-in}(\kappa, \text{“stop”})\rangle \circ \epsilon\}\rangle\!\rangle$$
$$\text{where } \textbf{true} = \text{task-cancelled}(\sigma, \ell)$$

$$\langle\!\langle \sigma_0, (\ell, \kappa) \circ Q, P \uplus \{\epsilon\}\rangle\!\rangle \qquad \text{[Asyncio-Schedule-Cancelled]}$$
$$\longrightarrow \langle\!\langle \sigma_1, Q, P \uplus \{\langle \ell, \textbf{throw-in}(\kappa, \text{“stop”})\rangle \circ \epsilon\}\rangle\!\rangle$$
$$\text{where } \textbf{true} = \text{task-cancelled}(\sigma_0, \ell),\ \sigma_1 = \text{task-uncancel}(\sigma_0, \ell)$$

$$\langle\!\langle \sigma, (\ell, \kappa) \circ Q, P \uplus \{\epsilon\}\rangle\!\rangle \qquad \text{[Tokio/Smol-Schedule-Cancelled]}$$
$$\longrightarrow \langle\!\langle \sigma, Q, P \uplus \{\langle \ell, e_{fail}\rangle \circ \epsilon\}\rangle\!\rangle$$
$$\text{where true} = \text{task-cancelled}(\sigma, \ell)$$
$$e_{fail} = \textit{task-set-failed}(\ell, ())\ ;\ \textbf{sys-start-soon}(\textit{task-get-dependents}(\ell))$$

AWARENESS DIRECTION PERSISTENCE

Fig. 16. An illustrative subset of cancellation semantics.

4.5 Seven Runtimes, One Program

Figure 17 brings the design dimensions and the formal model full circle: it evaluates program `ex2` from Figure 1 under all seven runtimes at once, branching precisely where their reduction rules diverge. Unlike the table of outcomes in Figure 1c, we can use the formal model to explain *why* a language produces the output it does. The branching evaluation highlights the fact that languages can produce the same output for different reasons. For example, C♯ and Swift both produce `AC`, however, C♯ does so because the runtime loop terminates running tasks and Swift does so because it cancels unawaited tasks at the end of their scope. In the evaluation context of `ex2`, these semantics produce the same output, but in a different context they would likely lead to different outputs. The branching evaluation also highlights that diverging branches can lead to the same output.

In sum, we presented an illustrative subset of our formal model, constructed as a series of language extensions atop the core λ_v (Figure 12c). For each rule—[Async-App], [Spawn], [Await], and the cancellation rules—we marked where the language variants diverge, coloring each fragment by the design dimension it decides. We use the formal model to explain diverging output of real-world examples, illustrated in Figure 17 for `ex2` from Figure 1. The included artifact provides the full Redex model that we hope can serve as common ground when communicating asynchronous semantics between language communities.

5 Related Work

The study of asynchronous programming sits at the intersection of several bodies of literature. We organize related work into three groups: foundational models of concurrency from which asynchrony draws its semantics, the development of deferred-value abstractions that gave asynchrony its first concrete programming primitives, and the functional and monadic tradition that supplied the compositional framework underlying async/await.

[7] To perform asynchronous cleanup work, runtimes like Trio and Swift will provide cancellation *shields*, which temporarily protect a task against cancellation so that async work can finish.

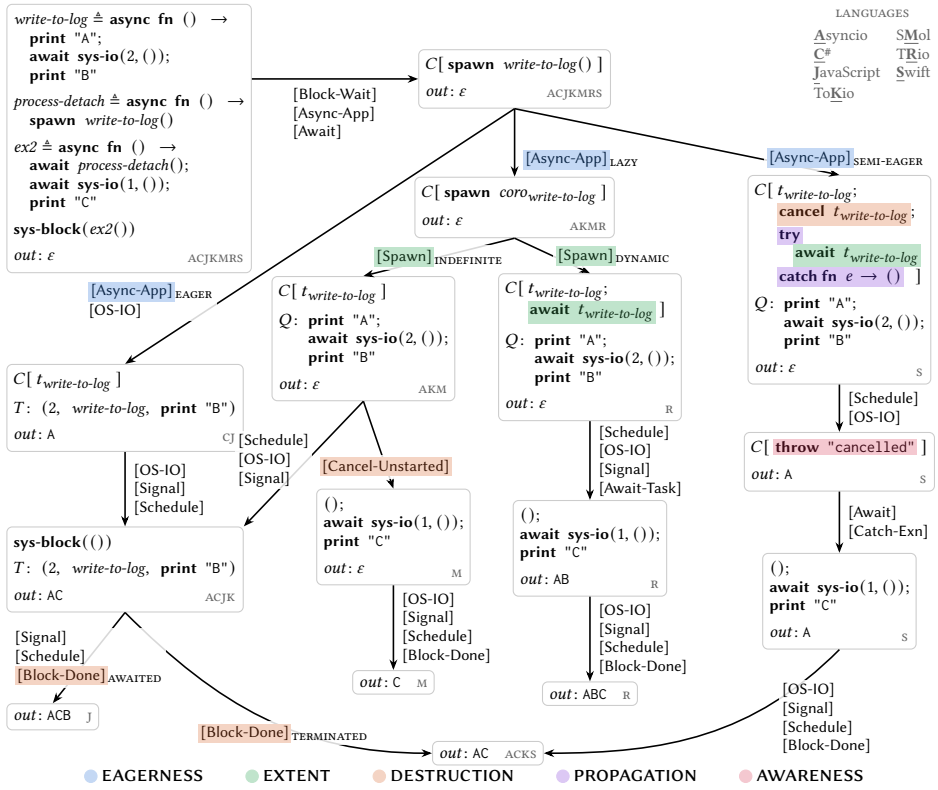


Fig. 17. The branching evaluation of program ex2 from Figure 1. Nodes represent the program state at that point, edges represent multiple reductions. Nodes have multiple outgoing edges when runtimes disagree on the next applicable rule. Highlighting indicates the respective design dimension; subscripts indicate each branch's design point. For example, on the rightmost path, Swift's SEMI-EAGER [Async-App] queues *write-to-log* and injects a scope-exit epilogue that cancels its dependency (DESTRUCTION), awaits it (EXTENT), and discards any resulting exception (PROPAGATION). When *write-to-log* runs, its cancellation-aware [OS-IO] implementation observes the flag and raises a *cancelled* exception (AWARENESS).

The earliest formal treatments of concurrency were not primarily concerned with asynchrony as we have defined it, but they established the semantic vocabulary on which later work depends. The first formalized models were the actor model [1, 3] and communicating sequential processes [19]. The π-calculus extended this work by allowing first-class channels that can also be sent between processes [30].

Deferred value abstractions have been introduced in many forms and under many aliases. Baker and Hewitt [3] introduced the term *future,* which was also used by Multilisp, which extended Scheme with several forms for expressing concurrency [16]. An independent exploration of demand-driven evaluation by Friedman and Wise [13] brought about the lazy *cons* operator. Miller et al. [29] defined *promise pipelining:* rather than blocking to receive a value, a program could register a continuation that would execute upon resolution.

Purer functional languages asked not how to represent a deferred value, but rather how to compose effectful computations while maintaining referential transparency. Jones et al. [22] built concurrent Haskell, which was later extended with shared-state concurrency [17]. The monadic

model influenced F♯, the first major async/await design, where the compiler automatically rewrote a block of code written in direct-style into its continuation-passing equivalent [44].

In addition, Bierman et al. [5] published a formal model for Featherweight C♯ (v5) from which we drew much inspiration. The term "structured concurrency" emerged from practitioners in early 2016 [41–43], but the idea of structured and automatic resource management was introduced in Racket's *custodians* [12].

6 Discussion

The book is not closed on the design of straight-line asynchrony. Communities are actively exploring extensions to the core async/await semantics presented in this work.

Elizarov et al. [10] use the *suspend* keyword in Kotlin to define async functions. Instead of an explicit await keyword, the language automatically introduces suspension points where necessary. They argue that this approach avoids forgotten awaits, and further that it improves the look-and-feel of calling a suspending function. However, this claim, and other consequences of this design, are not accompanied by an evaluation.

The Zig programming language is stabilizing its "colorblind" async/await [6, 23]. Functions are parameterized over an I/O implementation; libraries exploit asynchrony when available, and clients pass whichever I/O implementation they have, including a synchronous variant. This approach facilitates sync–async interop, but introduces new risks where libraries written explicitly for asynchrony deadlock or panic at runtime if asynchrony is unavailable.

C++20 provides a flexible implementation of coroutines that does not ascribe a semantics to its async/await [47]. We cannot attribute C++ to any particular design point in the taxonomy provided in Table 1 because each axis is configurable. Although elegant and neutral, the choice of full programmability makes each library an async DSL; knowledge transfer between projects within the same language becomes exceedingly difficult.

Where do we go from here? We set out to examine the space of straight-line asynchrony, ourselves confused by some statements we read in individual language descriptions. In particular, as we note in Section 2.1, these languages motivate async from a perspective of similarity to synchrony, which we found difficult to reconcile. Our design space exploration demonstrates the numerous ways in which this analogy fails.

The principal use for our exploration is to help developers. Developers transitioning from one language to another carry their prior knowledge to make sense of the new in terms of the old. It is especially confusing when the new language has the same keywords and similar description, but different semantics. We believe that our articulation of the design space, such as Table 1, can help learners and educators talk more effectively about how to translate knowledge between languages. We also believe that our exploration can help language designers build new features for straight-line asynchrony. Our design space can help enumerate key design dimensions, preventing important details from slipping through the cracks. It can also help designers more effectively compare and contrast their design against related languages.

An open challenge is to build a body of empirical data that can support designers in reasoning about the trade-offs of different approaches. For example, how much overhead does SEMI-EAGER async function application incur versus EAGER, and how much more concurrency does it gain? To what extent does SIMULTANEOUS cancellation prevent realistic bugs that would occur with a BOTTOM-UP cancellation strategy? Which of these semantics best matches developer expectations for the behavior of asynchronous code? Which decisions lead to more errors? We hope that this exploration can inspire practitioners and academics alike to build a better understanding of the evolving world of asynchronous programming.

Data-Availability Statement

The software artifact for this paper is publicly available [15] and contains three main components:

(1) All the code associated with the examples in the paper, runnable in their respective languages.
(2) The full suite of Redex models providing the described semantics.
(3) A differential fuzzer to test each model against its real-world counterpart.

Acknowledgments

We want to thank the many people who provided feedback on this work: Luke Wagner and Alex Crichton for detailing the WASM component model, John McCall for helping us understand the nuances of Swift's semantics, Akshay Narayan and Malte Schwarzkopf for their close readings of the text and thoughtful feedback that helped scope and frame the paper, and the members of the Brown PLT and Cognitive Engineering Lab groups for testing artifacts and spontaneously answering the question *What do you think this program prints?*.

This work is partially supported by US NSF grant CCF-2227863 and by the DARPA under Agreement No. HR00112420354. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not reflect the views of our funders.

References

[1] Gul Agha. 1986. *Actors: A Model of Concurrent Computation in Distributed Systems.* The MIT Press. doi:10.7551/mitpress/1086.001.0001
[2] Jake Archibald. 2023. The gotcha of unhandled promise rejections. https://jakearchibald.com/2023/unhandled-rejections/
[3] Henry C. Baker and Carl Hewitt. 1977. The incremental garbage collection of processes. In *Proceedings of the 1977 symposium on Artificial intelligence and programming languages.* Association for Computing Machinery, New York, NY, USA, 55–59. doi:10.1145/800228.806932
[4] Jacob Bartlett. 2023. Why is my Task running on the main thread? https://blog.jacobstechtavern.com/p/why-is-task-running-on-main-thread
[5] Gavin Bierman, Claudio Russo, Geoffrey Mainland, Erik Meijer, and Mads Torgersen. 2012. Pause 'n' Play: Formalizing Asynchronous C#. In *ECOOP 2012 – Object-Oriented Programming.* Vol. 7313. Springer Berlin Heidelberg, Berlin, Heidelberg, 233–257. doi:10.1007/978-3-642-31057-7_12
[6] Loris Cro. 2020. What is Zig's "Colorblind" Async/Await? https://kristoff.it/blog/zig-colorblind-async-await/
[7] M. a m a D. 2025. Why only creating a task will run the coroutine in python? https://stackoverflow.com/q/79755548
[8] Olivier Danvy and Andrzej Filinski. 1989. A functional abstraction of typed contexts. BRICS 89/12.
[9] ECMA Ecma. 2025. ECMAScript® 2026 Language Specification. https://262.ecma-international.org/16.0
[10] Roman Elizarov, Mikhail Belyaev, Marat Akhin, and Ilmir Usmanov. 2021. Kotlin coroutines: design and implementation. In *Proceedings of the 2021 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software.* ACM, Chicago IL USA, 68–84. doi:10.1145/3486607.3486751
[11] Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. 2009. *Semantics Engineering with PLT Redex* (1st ed.). The MIT Press.
[12] Matthew Flatt, Robert Bruce Findler, Shriram Krishnamurthi, and Matthias Felleisen. 1999. Programming languages as operating systems (or revenge of the son of the Lisp machine). *SIGPLAN Not.* 34, 9 (Sept. 1999), 138–147. doi:10.1145/317765.317793
[13] Daniel P. Friedman and David S. Wise. 1976. CONS Should Not Evaluate Its Arguments. In *Proceedings of the Third International Colloquium on Automata, Languages and Programming (ICALP 1976).* Edinburgh University Press, Edinburgh, Scotland, 257–284.
[14] Al Gebra. 2023. How can I start a Python async coroutine eagerly? https://stackoverflow.com/q/69145007
[15] Gavin Gray. 2026. *Artifact: A Design Space Exploration of Async/Await (v1.3).* doi:10.5281/zenodo.21765917
[16] Robert H. Halstead. 1985. MULTILISP: a language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst.* 7, 4 (Oct. 1985), 501–538. doi:10.1145/4472.4478
[17] Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. 2005. Composable memory transactions. In *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Chicago, IL, USA) *(PPoPP '05).* Association for Computing Machinery, New York, NY, USA, 48–60. doi:10.1145/1065944.1065952

[18] Alexander Hartl. 2022. asyncio: Use strong references for free-flying tasks · Issue #91887 · python/cpython. https://github.com/python/cpython/issues/91887

[19] C. A. R. Hoare. 1978. Communicating sequential processes. *Commun. ACM* 21, 8 (Aug. 1978), 666–677. doi:10.1145/359576.359585

[20] Apple Inc. 2025. Embracing Swift concurrency - WWDC25 - Videos. https://developer.apple.com/videos/play/wwdc2025/268/

[21] Apple Inc. and the Swift Project Authors. 2026. Calling Asynchrnous Functions in Parallel. https://docs.swift.org/swift-book/documentation/the-swift-programming-language/concurrency/#Calling-Asynchronous-Functions-in-Parallel

[22] Simon Peyton Jones, Andy Gordon, and Sigbjorn Finne. 1996. Concurrent Haskell. 295–308. https://www.microsoft.com/en-us/research/publication/concurrent-haskell/

[23] Andrew Kelley. 2025. Zig's New Async I/O (Text Version) - Andrew Kelley. https://andrewkelley.me/post/zig-new-async-io-text-version.html

[24] Loris Lacuna. 2025. asyncio: a library with too many sharp corners. https://sailor.li/asyncio.html

[25] Kuang-Chen Lu and Shriram Krishnamurthi. 2024. Identifying and Correcting Programming Language Behavior Misconceptions. *Proceedings of the ACM on Programming Languages* 8, OOPSLA1 (April 2024), 334–361. doi:10.1145/3649823

[26] MaPePeR. 2023. Starting or resuming async method/coroutine without awaiting it in Python. https://stackoverflow.com/q/77667669

[27] John McCall, Doug Gregor, and Konrad Malawski. 2021. Swift Evoluation Proposals — 0317: async let. https://github.com/swiftlang/swift-evolution/blob/main/proposals/0317-async-let.md

[28] John McCall, Joe Groff, Doug Gregor, and Konrad Malawski. 2021. Swift Evolution Proposals —0304: Structured Concurrency. https://github.com/swiftlang/swift-evolution/blob/main/proposals/0304-structured-concurrency.md

[29] Mark S. Miller, E. Dean Tribble, and Jonathan Shapiro. 2005. Concurrency Among Strangers. In *Trustworthy Global Computing*. Vol. 3705. Springer Berlin Heidelberg, Berlin, Heidelberg, 195–229. Series Title: Lecture Notes in Computer Science. doi:10.1007/11580850_12

[30] Robin Milner, Joachim Parrow, and David Walker. 1992. A calculus of mobile processes, I. *Information and Computation* 100, 1 (Sept. 1992), 1–40. doi:10.1016/0890-5401(92)90008-4

[31] Ana Lúcia De Moura and Roberto Ierusalimschy. 2009. Revisiting coroutines. *ACM Transactions on Programming Languages and Systems* 31, 2 (Feb. 2009), 1–31. doi:10.1145/1462166.1462167

[32] David Pacheco. 2025. 397 - Challenges with async/await in the control plane / RFD / Oxide. https://rfd.shared.oxide.computer/rfd/397

[33] Rain Paharia. 2025. 400 - Dealing with cancel safety in async Rust / RFD / Oxide. https://rfd.shared.oxide.computer/rfd/400

[34] Rain Paharia. 2025. Cancelling async Rust · sunshowers. https://sunshowers.io/posts/cancelling-async-rust/

[35] Embassy project contributors. 2019. Embassy. https://github.com/embassy-rs/embassy.

[36] Saad. 2021. Using async/await with a forEach loop. https://stackoverflow.com/q/37576685

[37] Yury Selivanov. 2015. PEP 492 – Coroutines with async and await syntax | peps.python.org. https://peps.python.org/pep-0492/

[38] Nathanial J Smith. 2016. Some thoughts on asynchronous API design in a post-async/await world — njs blog. https://vorpus.org/blog/some-thoughts-on-asynchronous-api-design-in-a-post-asyncawait-world/

[39] Nathanial J Smith. 2018. Notes on structured concurrency, or: Go statement considered harmful. https://vorpus.org/blog/notes-on-structured-concurrency-or-go-statement-considered-harmful/

[40] ECMA C# standard committee. 2022. C# Language Specification. Last updated: June 13, 2026. https://learn.microsoft.com/en-us/dotnet/csharp/specification

[41] Martin Sustrik. 2016. Getting rid of state machines (I). https://sustrik.github.io/250bpm/blog:69/

[42] Martin Sustrik. 2016. Getting rid of state machines (II). https://sustrik.github.io/250bpm/blog:70/

[43] Martin Sustrik. 2025. Structured Concurrency. https://www.250bpm.com/p/structured-concurrency

[44] Don Syme, Tomas Petricek, and Dmitry Lomov. 2011. The F# Asynchronous Programming Model. In *Practical Aspects of Declarative Languages (PADL 2011) (Lecture Notes in Computer Science, Vol. 6539)*. Springer, 175–189. doi:10.1007/978-3-642-18378-2_15

[45] Mads Torgersen. 2010. Asynchronous Programming in C# and Visual Basic. (Oct. 2010).

[46] Bill Wagner. 2025. Asynchronous programming - C#. https://learn.microsoft.com/en-us/dotnet/csharp/asynchronous-programming/

[47] ISO/IEC JTC1 SC22 WG21. 2017. Technical Specification for {C++} Extensions for Coroutines. https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/n4680.pdf

[48] William. 2019. How does javascript async/await actually work? https://stackoverflow.com/q/57160341

[49] Grant Winney. 2021. Using Async, Await, and Task to keep the WinForms UI responsive. Section: posts. https://grantwinney.com/using-async-await-and-task-to-keep-the-winforms-ui-more-responsive/
[50] withoutboats. 2018. 2394 — async/await - The Rust RFC Book. https://rust-lang.github.io/rfcs/2394-async_await.html
[51] withoutboats. 2023. Why async Rust? https://without.boats/blog/why-async-rust/

Received 2026-03-17; accepted 2026-06-10