Spicing Up Dart with Side Effects

A set of extensions to the Dart programming language, designed to support asynchrony and generator functions

Erik Meijer, Applied Duality
Kevin Millikin, Google
Gilad Bracha, Google

The Dart programming language has recently incorporated a set of extensions designed to support asynchrony and generator functions. Because Dart is a language for Web programming, latency is an important concern. To avoid blocking, developers must make methods asynchronous when computing their results requires nontrivial time. Generator functions ease the task of computing iterable sequences.

The extensions described in this article provide synchronous functions in two flavors: normal functions that produce a single result; and generator functions that produce a sequence of results. It follows, for reasons of symmetry, that asynchronous functions should come in two flavors as well: those that produce a single result asynchronously; and generators that produce a sequence of results asynchronously. In the design described here, normal functions/generators and asynchrony/synchrony are orthogonal dimensions. Table 1 illustrates these dimensions and shows the types involved.

	One	Many
Sync	T	Iterable<T>
Async	Future<T>	Stream<T>

TABLE 1 Generators and Synchrony: Orthogonal dimensions

Unfortunately, control structures in contemporary languages are optimized for simple synchronous calls, and as soon as the need arises to compose asynchronous methods, developers are left to their own devices, forced to write explicit CPS (continuation passing style) code by hand as if they were human compilers. Matters are even worse for code that manipulates asynchronous streams, but even creating synchronous iterables involves tedious manual construction of CPS state machines.

Fortunately, some modern language implementations already compile using continuations, which opens up the opportunity to perform the necessary CPS transformation automatically on the programmer's behalf. By simply marking method bodies as **async**, **async***, or **sync***, developers can write vanilla imperative control structures such as loops, conditionals, **try/catch/finally**,

and break/continue statements to escape "callback hell" as the compiler takes care of all the heavy lifting.

ASYNCHRONOUS PROGRAMMING USING FUTURES

In Dart and many other languages, methods that asynchronously return a single result do this by immediately returning a value of type Future<T>. The .then(f, onError: g) method on Future registers continuation callbacks for when the future completes successfully with a value f of type T or, respectively, with an exception g. The reward for acknowledging asynchrony by using futures is "callback hell." Developers are not able to use regular control-flow constructs anymore and instead need to transform the code manually into CPS.

Imagine for a moment that Dart had a synchronous API for making HTTP requests. Then we could write the following straightforward code to create a new client, contact example.com to perform a search, print the result, and close the connection:

```
getPage(t) {
   var c = new http.Client();
   try {
      var r = c.get('http://example.com/search?q=$t');
      print(r);
   } finally {
      return c.close();
   }
}
```

Of course, doing a blocking synchronous HTTP request is not the best idea, which is why in Dart all network APIs are asynchronous and return futures. But instead of using ; and try-finally, we must use .then and .whenComplete, and as a consequence the high-level structure of the code is destroyed:

```
Future getPage(t) {
   var c = new http.Client();
   return c.get('http://example.com/search?q=$t')
      .then((r) {
         print(r);
      }).whenComplete(() { return c.close(); });
}
```

This is where async functions enter the picture. These are functions that immediately, and automatically, return a future when called. The future is

later completed in a way determined by the function body. If the function returns successfully, then the future is completed with the value computed by the function body. If the function throws an exception, then the future is completed with the object thrown.

With `async` functions, we can write fully asynchronous code, using regular control structures, very much like the imaginary blocking synchronous code we started with:

```
Future getPage(t) async {
   var c = new http.Client();
   try {
      var r = await c.get('http://example.com/search?q=$t');
      print(r);
   } finally {
      await c.close();
   }
}
```

The `async` body shown here uses the `await` expression to suspend execution of the function until the future that is being awaited completes. Then, if the future completed with a value `r`, the `await` evaluates to `r`. If the expression that is awaited evaluates to a value that is not a future or throws an exception, the result is wrapped in a future before being processed further by the `await`.

Futures represent computations that eventually will complete with either a value or an exception. In Dart, we can register callbacks `onValue(T value)` and `onError(Exception e)` for those two events using the `.then` method on `Future<T>`. To create a future from scratch, Dart provides the factory type `Completer<T>` (in other languages such as Scala, completers are often called *promises*). Given a completer, we can tear off a future via the `.future` property, and that future can be completed (at most once) with a value or an error via the `.complete(T value)` or `.completeError(Object e)` methods.

Note that `await e` is just a normal expression inside an `async` function and thus can appear anywhere inside the body where an expression can appear, including inside `catch` and `finally` blocks. The `async` modifier can also be applied to the bodies of closures and getters but not on constructors or setters. As a result, an `async` function can be written in a way that is very close to how synchronous functions are written, with familiar sequential control constructs: loops, conditionals, and `try-catch-finally`.

While **async** functions may remind us of the **async/await** feature in C#, note that there are differences. In C#, an **async** function executes *synchronously* from when it is called until the first **await** expression is encountered. Only if the value has not completed synchronously, or has thrown an exception, is the computation awaited. Otherwise, execution continues using the value of the completed future. As a result, an asynchronous function may be executed totally synchronously, even though marked as **async**. Another difference with C# is that in Dart it is the function *body* that is marked **async**. This emphasizes that asynchrony is a property of the function's implementation, and the signature of a function is not affected. Furthermore, in Dart, await always suspends execution, even if the result of the expression being awaited is not a future. Finally, in C# control cannot escape **finally** blocks, whereas in Dart, as in Java, **finally** blocks can transfer control back to wherever we want. This makes the C# translation of **async/await** that retains the original program structure unfeasible in Dart, and this is what triggered the continuation-based implementation.

The **async** functions in Facebook's new Hack language are similar to C#'s **async** methods and eagerly execute the body of an **async** function until the first **await**; already completed futures are not awaited.

Conceptually, the semantics of an **await** expression `x = await e; rest[x]` is `e.then((r){ x = r; rest[x] });` of course, one of the subtleties is what exactly the "rest of the computation" means, especially when the **await** expression appears inside a **try-catch-finally** statement or a loop, as in the following (artificial) example:

```
foo() async {
   var x;
   while(true) {
      try { x = await foo(); return x+1; } finally { continue; }
   }
}
```

To show what **async** functions mean, we define a Scott-Strachey style[5] continuation semantics for a "featherweight" subset of Dart methods defined by the following grammar:

```
expression ::= identifier = expression
             | expression.identifier(expression)
             | await expression

statement ::= { … statement … }
            | expression;
            | return expression;
            | while (expression) statement
            | try statement finally statement

method ::= identifier(identifier) async {
              var identifier; … statement …
           }
```

Without loss of generality, we assume that expressions and statements will always terminate successfully (i.e., no exceptions are thrown). Furthermore, to avoid clutter, we assume that methods have a single argument, method bodies have a single local variable, and all calls take a receiver and a single argument.

Don't be intimidated by the Greek symbols and higher-order functions. The semantic rules are based on the operational intuition of how Dart **async** methods are executed and match the declarative description of the official Dart language specification.[2]

The semantics for statements $\mathcal{C}$ ⟦s⟧ (ρ,σ) takes two callback functions (or *continuations*) named ρ and σ. The return continuation ρ denotes what should happen if the statement **s** executes **return;** the success continuation σ denotes what should happen when execution of **s** falls off the end without encountering a **return**. Hence both continuations have the same type, i.e., they take any value and return **void**.

The semantics for expressions $\mathcal{E}$ ⟦e⟧ (σ) needs only a success continuation σ since **return** is a statement and hence one cannot **return** from a method inside an expression.

The semantics for method declarations $\mathcal{M}$ ⟦d⟧ wires everything up by allocating a new **Completer** that is completed by the initial return and success continuations of the method body, and whose future is returned immediately by the generated wrapper function. The method body itself is run in a new future.

Starting with the semantics of a **return** statement, the rule $\mathcal{C}$ ⟦**return e;**⟧ (ρ,σ) specifies that first the expression **e** is evaluated to obtain a value **r**, as indicated by the call $\mathcal{E}$ ⟦e⟧ (λr→…), and then the value of the expression

r is passed to the return continuation of the return statement, as indicated by ρ(r). The complete rule then reads as follows:

$\mathcal{C}$ ⟦return e;⟧ (ρ,σ) = $\mathcal{E}$ ⟦e⟧ (λr→ρ(r))

Of course, for async methods the most interesting case is the semantics of await expressions $\mathcal{E}$ ⟦await e⟧ (σ). The semantics formalizes that first the expression $\mathcal{E}$ ⟦e⟧ (λr→…) is evaluated to obtain a future r and then, when r is completed, r.then(σ) will continue the evaluation of the rest of the program σ:

$\mathcal{E}$ ⟦await e⟧ (σ) = $\mathcal{E}$ ⟦e⟧ (λr→r.then(σ))

This is exactly what is to be expected when "awaiting" a future. Remember that for simplicity we don't deal with failure; if we did, the semantics would simply take an additional exception continuation ε, which would be passed as the error continuation of the future. The full semantics for async methods reads as follows:

```
E ⟦x=e⟧ (σ) = E ⟦e⟧ (λr→{ x=r; σ(r); })
E ⟦a.f(b)⟧ (σ) = E ⟦a⟧ (λa→E ⟦b⟧ (λb→σ(a.f(b)))
E ⟦await e⟧ (σ) = E ⟦e⟧ (λr→r.then(σ))

C ⟦{ s1 s2 }⟧ (ρ,σ) = C ⟦s1⟧ (ρ, λr→C ⟦s2⟧ (ρ,σ))
C ⟦e;⟧ (ρ,σ) = E ⟦e⟧ (σ)
C ⟦return e;⟧ (ρ,σ) = E ⟦e⟧ (λr→ρ(r))
C ⟦while(e) s⟧ (ρ,σ) = loop()
   where loop() = E ⟦e⟧ (λr→{
            if(r){ C ⟦s⟧ (ρ, λr→loop()); } else { σ(null); }
         })
C ⟦try s1 finally s2⟧ (ρ,σ) =
   C ⟦s1⟧ (λr→C ⟦s2⟧ (ρ,λs→ρ(r)), λr→C ⟦s2⟧ (ρ,λs→σ(null)))

M ⟦f(a) async { var x; s }⟧ = f(a) {
   var result = new Completer();
   new Future(λ()→{
      var x;
      C ⟦s⟧ (λr→result.complete(r), λr→result.complete(null))
   });
   return result.future
}
```

In the following example, applying the semantics to the async method on the left will result in the regular method on the right that uses .then and

recursion to implement the sequencing and looping of the original **async** method:

```
f() async {
   var x;
   while(await g()) {
     x = await h();
  }
  return x;
}
```

```
f() {
  var result = new Completer();
  new Future(λ()→{
     var x;
     loop() {
        g().then(λs→
        if(s) {
           h().then(λs→{ x = s; loop(); });
        } else {
           result.complete(x);
        });
     }
     loop();
  });
  return result.future;
}
```

A reassuring property of the abstract continuation semantics is that the resulting code is close to what a developer would have written by hand to achieve the same effect.

As noted, the semantics described here necessarily paints a simplified picture. To deal with the full Dart language and not just the featherweight subset, the Dart2Dart compiler carries around a couple of additional continuations to deal with **break** and **continue** statements, and **switch** and **catch** blocks. The compiler also defunctionalizes continuations back into direct style as much as possible (i.e., statements that contain no await expressions are compiled back down to themselves) and removes unnecessary "administrative" applications of continuations. Fundamentally, however, the compiler works the same as the semantics, except that it adds all the gory details required to move beyond the featherweight language category. To learn more about compiling with continuations, see the blog post by Matthew Might (http://matt.might.net/articles/cps-conversion/) and the references therein.

The Dart VM (virtual machine) has direct access to the stack, return addresses, and exception tables of the runtime and hence is able to implement the language enhancements described here at a lower level of abstraction. These runtime structures are concrete representations of the various continuations that the denotational semantics manipulates. To learn

more about describing the semantics of **async**, **sync***, and **async*** on the level of stacks and return addresses, see the following papers: http://dl.acm.org/citation.cfm?id=2367181 and http://dl.acm.org/citation.cfm?id=1297063.

ITERATORS AND ITERABLES

Now that we have explained how **async** methods simplify asynchronous code that returns a single value, let's shift the focus to bulk processing of collections. The **Iterable** and **Iterator** interfaces (or slight variations thereof) are the workhorses of collection libraries in almost every modern object-oriented language. Consuming iterables is deceptively easy because of the **foreach** loop:

```
Iterable xs = [1, 2, 3, 4, 5];
for(x in xs) { print(x); }
```

What really happens is that **for(x in xs)** hides the boilerplate of getting a fresh iterator from the iterable **xs** and iterating over that. In other words, a **for** loop in Dart is syntactic sugar for the following **while** loop:

```
Iterable xs = [1, 2, 3, 4, 5];
var _xs = xs.iterator;
while(_xs.moveNext()){ var x = _xs.current; print(x); }
```

Dart, however, has no syntactic support for *producing* iterables. This requires developers to perform the same cruel and unnatural acts that they had to do when writing asynchronous methods without the help of **async** functions. As an example, let's attempt to write the standard library function **filter** from scratch. Given a predicate, **filter** must return a new iterable where all values for which the predicate is **false** are removed:

```
Iterable filter(Iterable src, predicate) {
   return new FilterIterable(src, predicate);
}
```

The type **FilterIterable** builds on **IterableBase**, which implements all methods of **Iterable** except **iterator**. The **iterator** method returns an instance of **FilterIterator** that will filter out values from the iterator.

```
class FilterIterable extends IterableBase {
   var src, predicate;
   FilterIterable(this.src, this.predicate);
   FilterIterator get iterator {
      return new FilterIterator(src.iterator, predicate);
   }
}
```

Now we can actually implement the logic for removing values:

```
class FilterIterator extends Iterator {
   var src, predicate;
   FilterIterator(this.src, this.predicate);
   bool moveNext() {
      while (src.moveNext()) {
        if (predicate(src.current)) { return true; }
      }
      return false;
   }
   get current { return src.current; }
}
```

That is a lot of boring ceremony when the only interesting line is `if(predicate(src.current)){ … produce the next value … }`. It is a little harder to see at the moment that this code is actually also a manual CPS transform, but that will become clear in a moment, when the continuation semantics for synchronous generators is defined.

Synchronous generator functions are sugar for defining iterators and are defined by marking their body with the `sync*` modifier. Generators immediately return an `Iterable` when called. When we subsequently get an iterator from the iterable and call `moveNext()` on it, the body is executed until it hits a `yield` or `return` internally. If it reaches a `return` (again with the caveat it does not get hijacked by a `finally` clause), the iterator is done, and further calls to `moveNext()` will return `false`. Otherwise, subsequent calls to `moveNext()` resume the function where it left off, until the next `yield` or `return`. With generators, the code for `filter` collapses to a one-liner:

```
Iterable filter(Iterable src, predicate) sync* {
   for(var s in src){ if(predicate(s)) { yield s; }}
}
```

A careful reading of the API documentation for `Future` reveals that nested futures are automatically flattened. For generators we generally want to maintain nested iterables, but in certain cases we want to splice a nested iterable into its parent iterable. Take the following example where the sequence `range(s,n) = s, s+1, …, s+n-1` is recursively generated:

```
Iterable range(s, n) sync* {
   if(n>0){ yield s; for(var i in range(s+1, n-1)) yield i; }
}
```

The problem with this implementation of `range` is that value `s+i` is being yielded `i` times, and hence the runtime complexity of `range` is quadratic.[4] This is not an artificial problem; if we concatenate two iterables by copying

```
Iterable append(Iterable left, Iterable right) sync* {
   for(var l in left){ yield l; } for(var r in right){ yield r; }
}
```

then we run into the same quadratic effect when appending lists in a left-associative manner, as in `append(append(xs,xs),xs)`. Implementing efficient list concatenation is nontrivial,[3] and the "quadratic append" problem pops up in many other contexts as well.

Instead of copying a nested iterable, Dart `sync*` methods allow us to splice a nested iterable into the result iterable by using the `yield*` statement, which can implement `range` thus:

```
range(s, n) sync* {
   if(n>0){ yield s; yield* range(s+1, n-1); }
}
```

The ability to implement splicing of nested iterables efficiently requires the result of a `sync*` function to be a Rose-tree that is traversed in depth-first order (using a stack) when we iterate over it. Formally, we use the fact that iterators are isomorphic to the recursive type of *resumptions*:

```
Iterator<T> ≅ μ Resumption.
            ()→(()U(T,Resumption)U(()→Resumption,Resumption))
```

A `Resumption` is a recursive function that unfolds an iterator either by terminating immediately, by returning a pair of a single value and a continuation resumption that will produce more values, or by returning a pair

of a nested iterable and a continuation resumption. Assume there is a function β (with inverse β^{-1}) that embeds `Iterable<T>` (i.e., iterator factories) into resumption factories of type `()→Resumption`. It ensures that both normal iterables and recursive calls to `sync*` methods are handled uniformly as resumptions.

The featherweight subset of Dart for `sync*` methods adds the `yield` and `yield*` statements and restricts `return` inside `sync*` methods to have no result expression. Again, without loss of generality, we ignore exceptions.

```
expression ::= identifier = expression
             |  expression.identifier(expression)

statement ::= { … statement … }
            |  expression;
            |  return;
            |  yield expression; | yield* expression;
            |  while(expression) statement
            |  try{ statement } finally statement

method ::= identifier(identifier) sync*{
              var identifier; … statement …
           }
```

The semantics for statements $\mathcal{C}$ ⟦s⟧ (ρ,σ) uses two continuations ρ and σ, since when executing `sync*` bodies we need to run `finally` blocks before completing the underlying iterable. The semantics for expressions $\mathcal{E}$ ⟦e⟧ (σ) remains pretty much as for `async` methods except that inside a `sync*` method we cannot `await` (for that we need `async*` methods, as explained later in this article). The semantics for method declarations $\mathcal{M}$ ⟦d⟧ wires everything up.

Return statements inside `sync*` methods don't have a result expression but have to go through the `finally` blocks like `return` statements in normal methods; hence, the semantics of a `return` statement $\mathcal{C}$ ⟦return;⟧ (ρ,σ) invokes the return continuation with an "end of resumption" value:

$\mathcal{C}$ ⟦return;⟧ (ρ,σ) = ρ()

For `sync*` methods the most interesting cases yield a single value $\mathcal{C}$ ⟦yield e;⟧ or a nested iterable $\mathcal{C}$ ⟦yield* e;⟧ . The semantics formalizes that first the expression $\mathcal{E}$ ⟦e⟧ (λr→…) is evaluated to obtain a result r, and then the execution of the `sync*` method returns a resumption pair of the value r and the rest of the computation σ:

$\mathcal{C}$ ⟦yield e;⟧ (ρ,σ) = $\mathcal{E}$ ⟦e⟧ (λr→(r,σ)))

When we yield* a nested iterable, r is coerced to a resumption using β so we do not have to worry about the difference between a recursive invocation of a sync* method or a regular method that returns an iterable.

```
𝒞 ⟦yield* e;⟧ (ρ,σ) = ℰ ⟦e⟧ (λr→(β(r),σ))
```

The full continuation semantics for sync* methods then reads as follows:

```
ℰ ⟦x=e⟧ (σ) = ℰ ⟦e⟧ (λr→{ x=r; σ(r); })
ℰ ⟦a.f(b)⟧ (σ) = ℰ ⟦a⟧ (λa→ℰ ⟦b⟧ (λb→σ(a.f(b)))

𝒞 ⟦s1 s2⟧ (ρ,σ) = 𝒞 ⟦s1⟧ (ρ, λr→𝒞 ⟦s2⟧ (ρ,σ))
𝒞 ⟦e;⟧ (ρ,σ) = ℰ ⟦e⟧ (λr→ρ(r))
𝒞 ⟦return;⟧ (ρ,σ) = ρ()
𝒞 ⟦yield e;⟧ (ρ,σ) = ℰ ⟦e⟧ (λr→(r,σ))
𝒞 ⟦yield* e;⟧ (ρ,σ) = ℰ ⟦e⟧ (λr→(β(r),σ))
𝒞 ⟦while(e) s⟧ (ρ,σ) = loop()
   where loop() = ℰ ⟦e⟧ (λr→{
            if(r){ 𝒞 ⟦s⟧ (ρ, λr→loop()); } else { σ(null); }
         })
𝒞 ⟦try s1 finally s2⟧ (ρ,σ) =
   𝒞 ⟦s1⟧ (λ()→𝒞 ⟦s2⟧ (ρ,λs→ρ()), λr→𝒞 ⟦s2⟧ (ρ,λs→σ(r)))

ℳ ⟦f(a) sync* { var x; s }⟧ = f(_a){
   return new _IterableBase(λ()→{
      var x; var a = _a; return 𝒞 ⟦s⟧ (λ()→(), λr→())
   });
}
where
   class _IterableBase extends IterableBase {
      var _resumption;
      _IterableBase(this._resumption);
      Iterator get iterator { return β⁻¹(_resumption()); }
   }
```

Applying the semantics to the program on the left below yields:

```
range(s,n) sync* {
   if(n>0) {
      yield s;
      yield* range(s+1,n-1);
   }
}
```

```
range(s,n)→new _IterableBase(λ()→{
   if(n>0) { return
      (s,
       λ()→(β(range(s+1, n-1)),λ()→())
      );
   }
   return ();
});
```

While sync* functions may remind us of iterators in C#, we must again stress that there are differences. In C#, depending on the specified return type of an iterator method that contains **yield return** or **yield break** statements, iterators can return either enumerables (iterables) or enumerators (iterators). Since types are optional in Dart, sync* functions always return iterables. In Dart, **yield**, **yield***, and **return** statements can appear anywhere a regular statement can appear, including in **finally** blocks. Another difference with Dart is that in C# lambda expressions *cannot* be iterators (because we cannot specify their return type), whereas in Dart methods, functions, and getters can be marked as sync*. Perhaps the most important improvement of Dart sync* methods over C#'s iterators is the inclusion of **yield*** to splice in nested iterables.

To Cancel or Not To Cancel Is the Question

An open design decision is whether or not Dart iterators should support a cancel or dispose method. For simple iterators that is no problem, but once the power of sync* methods is introduced, it is easy to leave a method hanging in-flight holding on to expensive resources. For example, calling **moveNext()** just once on the following sync* method will return the first line of the file but never close it:

```
Iterable readLines(name) sync* {
   var file = new File(name);
   try {
      yield file.readNext();
   } finally {
      file.close();
   }
```

```
}
```

In C# when we **dispose** (cancel) an iterable, all **finally** blocks are run, and the **foreach** loop is surrounded by a **try/finally** block to ensure that the enumerable is always disposed of after use. Adding this feature to **sync*** methods can be easily achieved by maintaining a "finally continuation" that when invoked will run all **finally** blocks and ignore any further **yield** and **yield*** statements encountered during cleanup.

ASYNC* = ASYNC+SYNC*

So far, we have seen how **async** and **sync*** methods simplify coding of methods that asynchronously return a single value and methods that synchronously return multiple values, respectively. Combining these two dimensions results in **async*** functions that simplify coding of methods that asynchronously produce streams of values. Just as an **async** function immediately returns a future, an **async*** function immediately returns a **Stream**. When (and only when) we listen to the stream, the function starts executing, just as a **sync*** method starts execution when **moveNext** is called on the **Iterable** it returns.

Say we want to scan a stream and incrementally emit the intermediate results of applying an asynchronous function to each incoming item. Using **async*** methods, this becomes just a matter of looping over the stream, awaiting the computation of the next state, and then yielding the current state:

```
Stream scan(Stream src, state, acc) async* {
   yield state;
   await for(var next in src) {
     try {
        yield (await accumulateAsync(state, next));
     } catch(e) {
        return; // swallow exception, don't do this at home!
     }
   }
}
```

As you can imagine, manually implementing this behavior using a **StreamController** is very tedious. We need to pause the source stream while awaiting the result of **accumulateAsync** and resume it when the future completes, and we need to handle a **try-catch** block around the **await**, cancellation of the resulting stream, etc. Thanks to **async*** methods, we can just write regular control-flow, and the Dart compiler will take care of handling all the difficult stream-management issues.

Dart streams are the asynchronous counterparts of iterables. Given a stream `ts` of type `Stream<T>`, we can listen to the stream for notification of new values by passing a callback `void onData(T event)`, and optionally `void onError(Exception e)` and `void onDone()`. The `listen` method then returns a `StreamSubscription` that has a method `cancel` to unsubscribe from the stream, plus `pause` and `resume` to control the speed of the producer. Streams are created either by transforming existing streams using operators such as `map` and `reduce` or by using a `StreamController` to which we can push values using the methods `add` and `addError`.

The featherweight subset of Dart that defines the semantics for `async*` methods is defined as follows:

```
expression ::= identifier = expression
            |  expression.identifier(expression)
            |  await expression

statement ::= statement statement
           |  expression;
           |  return;
           |  yield expression; | yield* expression;
           |  while(expression) statement
           |  try statement finally statement
           |  await for(var identifier in expression) statement

method ::= identifier(identifier) async* {
              var identifier; … statement …
           }
```

To keeps things simple, we will again ignore failure and not implement cancellation. Because `await for` loops can be nested, the semantics for statements in a featherweight `async*` body $\mathcal{C}[\![s]\!](\xi,\rho,\sigma)$ takes not only two continuations ρ and σ, but also a subscription ξ to the stream of the enclosing method or `await for` loop (for convenience, we often refer to ξ simply as "the immediately enclosing stream"). The immediately enclosing stream is needed so that when the result stream of the `async*` method is paused, we can `pause/resume` the body at a `yield` or `yield*` statement or when iterating over a nested stream using `await for`. The enclosing stream is initialized with `never`, which never emits a value, and hence `never.pause()` and `never.resume()` are no-ops and can be dropped.

The semantics for expressions $\mathcal{E}$ ⟦e⟧ (ξ,σ) also gets the subscription to the enclosing stream so it can pause/resume it when executing an await expression. The semantics for method declarations $\mathcal{M}$ ⟦d⟧ wires everything up by creating a subclass of stream controller whose .stream is returned immediately and whose .sink is used by the body of the async* method to yield values.

For expressions $\mathcal{E}$ ⟦await e⟧ (ξ,σ), first evaluate the expression $\mathcal{E}$ ⟦e⟧ (ξ,λr→{…}) using the enclosing stream ξ to yield a value r. Before awaiting r, pause ξ so that no values are pushed while waiting for the future to complete. Before the success continuation is invoked with the result v of the future, resume the enclosing stream ξ:

```
ℰ ⟦await e⟧ (ξ,σ) = ℰ ⟦e⟧ (ξ,λr→{
   ξ.pause(); r.then(λv→{ ξ.resume(); σ(v); });
})
```

Return statements inside async* methods are similar to sync* returns in that they invoke the return continuation to run all outstanding finally blocks and then close the result stream. Before yielding values, an async* method checks if its result stream, captured in the global variable result, has a pending pause request. If so, the immediately enclosing stream ξ is paused, then the result stream itself is paused by setting the onResume callback to continue when it is resumed by the consumer:

```
𝒞 ⟦yield e;⟧ (ξ,ρ,σ) = ℰ ⟦e⟧ (ξ,λr→{
   if(result.isPaused){
      ξ.pause(); result._onResume = λ()→{
         ξ.resume(); result.add(r); σ(null);
      };
   } else {
     result.add(r); σ(null);
   }
})
```

A similar pause check is performed before closing the result stream in the initial return and success continuation when $\mathcal{M}$ ⟦⟧ is invoked on the async* body. The most interesting case for async* methods is iterating over a stream 𝒞 ⟦await for(var x in e){s}⟧ (ξ,ρ,σ). First evaluate the loop expression ℰ ⟦e⟧ (ξ,λr→{…}) to obtain a stream r. Then pause the immediately enclosing stream ξ and listen to r instead to get the subscription _ξ. The onData callback executes the loop body in the context of _ξ, and the onDone callback restores ξ before invoking the success continuation:

```
𝒞 ⟦await for(var x in e) s⟧ (ξ,ρ,σ) = ℰ ⟦e⟧ (ξ,λr→{
   ξ.pause();
   var _ξ; _ξ = r.listen(λv→{
      var x = v;
      𝒞 ⟦s⟧ (_ξ, λ()→{ _ξ.close(); ξ.resume(); ρ(); }, λ()→{})
   }, onDone: λ()→{ _ξ.close(); ξ.resume(); σ(null); });
})
```

As with ordinary for loops over iterables, a new loop variable is allocated for each iteration.

The full continuation semantics for async* methods looks as follows:

```
ℰ ⟦x=e⟧ (ξ,σ) = ℰ ⟦e⟧ (ξ,λr→{ x=r; σ(r); })
ℰ ⟦a.f(b)⟧ (ξ,σ) = ℰ ⟦a⟧ (ξ,λa→ℰ ⟦b⟧ (ξ,λb→σ(a.f(b)))
ℰ ⟦await e⟧ (ξ,σ) = ℰ ⟦e⟧ (ξ,λr→{
   ξ.pause(); r.then(λv→{ ξ.resume(); σ(v); });
})

𝒞 ⟦{ s1 s2 }⟧ (ξ,ρ,σ) = 𝒞 ⟦s1⟧ (ξ,ρ,λr→𝒞 ⟦s2⟧ (ξ,ρ,σ))
𝒞 ⟦e;⟧ (ξ,ρ,σ) = ℰ ⟦e⟧ (ξ,λr→ρ(r))
𝒞 ⟦return;⟧ (ξ,ρ,σ) = ρ()
𝒞 ⟦yield e;⟧ (ξ,ρ,σ) = ℰ ⟦e⟧ (ξ,λr→{
   if(result.isPaused){
      ξ.pause(); result._onResume = λ()→{
         ξ.resume(); result.add(r); σ(null);
      };
   } else {
     result.add(r); σ(null);
   }
})
𝒞 ⟦yield* e;⟧ (ξ,ρ,σ) = ℰ ⟦e⟧ (ξ,λr→{
   if(result.isPaused){
      ξ.pause();
      result._onResume = λ()→{
        ξ.resume(); result.addStream(r); σ(null); };
   } else {
      result.addStream(r); σ(null);
   }
})
𝒞 ⟦while(e) s⟧ (ξ,ρ,σ) = loop()
   where loop() = ℰ ⟦e⟧ (ξ,λr→ {
```

```
            if(r){ 𝒞 ⟦s⟧ (ξ,ρ,λr→loop()); } else { σ(null); }
         })
𝒞 ⟦try s1 finally s2⟧ (ξ,ρ,σ) =
   𝒞 ⟦s1⟧ (ξ,λ()→𝒞 ⟦s2⟧ (ξ,ρ,λs→ρ()), λr→𝒞 ⟦s2⟧ (ξ,ρ,λs→σ(r)))
𝒞 ⟦await for(var x in e) s⟧ (ξ,ρ,σ) = ℰ ⟦e⟧ (ξ,λr→{
   ξ.pause();
   var _ξ; _ξ = r.listen(λv→{
      var x = v;
      𝒞 ⟦s⟧ (_ξ,λ()→{ _ξ.close(); ξ.resume(); ρ(); }, λ()→{})
   }, onDone: λ()→{ _ξ.close(); ξ.resume(); σ(null); });
})

ℳ ⟦f(a) async* { var x; s }⟧ = f(_a){
   var result = new _StreamController(
   onListen: λ()→{
      var x; var a = _a;
      𝒞 ⟦s⟧ (never,
       λ()→{
          if(result.isPaused) {
            result._onResume = λ()→result.close();
          } else {
            result.close();
          }
       }, λr→ {
          if(result.isPaused) {
            result._onResume = λ()→result.close();
          } else {
            result.close();
          }
       })
   },
   onResume: λ()→{ result._onResume(); }
   );
   return result.stream;
}
```

Below, applying the semantics to the program on the left yields the code on the right:

```
Stream repeat(s) async* {          Stream repeat(_s) {
   while(true) {                      var result = new _StreamController(
```

```
      yield* s();
   }
}
```

```
   onListen: λ()→{
      var s = _s;
      loop() = {
         if(result.isPaused){
            result._onResume = λ()→{
               result.addStream(s());
               loop();
            };
         } else {
            result.addStream(s());
            loop();
         }
      }
      loop();
   },
   onResume: λ()→{
      result._onResume();
   });
   return result.stream;
}
```

async* methods are a novel feature, and as far as we know no other language supports them, although it has been suggested (see http://hendryluk.wordpress.com/2012/02/28/reactive-extensions-and-asyncawait/) as an extension for supporting the imperative creation on observable streams in C#. Jafar Hussain from Netflix has proposed a similar feature for ECMAScript, but so far ECMAScript lacks a standard type for asynchronous streams.

CONCLUSION

Most mainstream (imperative) languages are optimized to express synchronous computations where the caller blocks until the callee returns a value. As the distance over which callers and callees have to exchange data increases (from the cache, disk, and network), the latency of calls increases by many orders of magnitude,[1] and hence it becomes crucial that programming languages directly support asynchronous calls. In an asynchronous call, the caller does not block waiting for the callee to return but instead passes a continuation that the callee invokes if and when it eventually produces a value.

By reviving the lost art of denotational semantics, we have shown how to take synchronous control-flow constructs such as loops, conditionals, and **try-catch-finally** blocks and reinterpret them in a natural way in the context of

asynchronous computations, as well as for producing synchronous and asynchronous streams of values. In the near future, every contemporary programming language will likely support asynchronous methods in one way or another, and the semantics used in this article may aid other language designers in adopting the approach to fit the idiosyncrasies of their particular languages.

REFERENCES

1. Bonér, J. 2012. Latency numbers every programmer should know; https://gist.github.com/jboner/2841832.
2. Dart Language Specification; https://www.dartlang.org/docs/spec/.
3. Hughes, R. J. M. 1986. A novel representation of lists and its application to the function "reverse"; *Information Processing Letters* 22 (3): 141-144; http://www.cs.tufts.edu/~nr/cs257/archive/john-hughes/lists.pdf.
4. Jacobs, B., Meijer, E., Piessens, F., Schulte, W. Iterators revisited: proof rules and implementation; http://research.microsoft.com/en-us/projects/specsharp/iterators.pdf.
5. Stoy, J. E. 1977. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. Cambridge, MA: MIT Press; http://dl.acm.org/citation.cfm?id=540155.

LOVE IT, HATE IT? LET US KNOW
FEEDBACK@QUEUE.ACM.ORG

ERIK MEIJER (emeijer@applied-duality.com) is the founder of Applied Duality and professor of big-data engineering at TUDelft. He is perhaps best known for his contributions to programming languages such as Haskell, C#, Visual Basic, Hack, and Dart, as well as his work on big-data technologies such as LINQ and the Rx Framework.
KEVIN MILLIKIN (kmillikin@google.com) is a software engineer at Google who has worked on the V8 JavaScript engine and the Dart virtual machine. He was one of the original developers of V8's Crankshaft compiler, the first JavaScript compiler to feature adaptive optimization. He is currently working on dart2js.
GILAD BRACHA (gbracha@google.com) is the creator of the Newspeak (http://bracha.org/Site/Newspeak.html) programming language and coauthor of the Java Language Specification (http://docs.oracle.com/javase/specs/). He is currently a software engineer at Google where he works on Dart (https://www.dartlang.org/).