Pause ’n’ play: Formalizing asynchronous C$^{\sharp\star}$

G. Bierman[1], C. Russo[1], G. Mainland[1], E. Meijer[2], and M. Torgersen[2]

[1] Microsoft Research {gmb,crusso,gmainlan}@microsoft.com
[2] Microsoft Corporation {emeijer,madst}@microsoft.com

Abstract. Writing applications that connect to external services and yet remain responsive and resource conscious is a difficult task. With the rise of web programming this has become a common problem. The solution lies in using asynchronous operations that separate issuing a request from waiting for its completion. However, doing so in common object-oriented languages is difficult and error prone. Asynchronous operations rely on callbacks, forcing the programmer to cede control. This inversion of control-flow impedes the use of structured control constructs, the staple of sequential code. In this paper, we describe the language support for asynchronous programming in the upcoming version of C$^\sharp$. The feature enables asynchronous programming using structured control constructs. Our main contribution is a precise mathematical description that is abstract (avoiding descriptions of compiler-generated state machines) and yet sufficiently concrete to allow important implementation properties to be identified and proved correct.

1 Introduction

Mainstream programmers are increasingly adopting asynchronous programming techniques once the preserve of hard-core systems programmers. This adoption is driven by a variety of reasons: hiding the latency of the network in distributed applications; maintaining the responsiveness of single-threaded applications or simply avoiding the resource cost of creating too many threads. To facilitate this programming style, operating systems and platforms have long provided non-blocking, asynchronous alternatives to possibly blocking, synchronous operations. While these have made asynchronous programming possible they have not made it easy.

The basic principle behind these asynchronous APIs is to decompose a synchronous operation that combines issuing the operation with a blocking wait for its completion, into a non-blocking initiation of the operation, that immediately returns control, and some mechanism for describing what to do with the operation’s result once it has completed. The latter is typically described by a callback—a method or function. The callback is often supplied with the initiation as an additional argument. Alternatively, the initiation can return a handle which the client can use to selectively register an asynchronous callback or (synchronously) wait for the operation’s result.

Whatever the mechanism, the difficulty with using these APIs is fundamentally this: to transform a particular synchronous call-site into an asynchronous call-site requires the programmer to represent the continuation of the original site as a callback. Moreover, for this callback to resume from where the synchronous call previously returned,

* Extended version

it must preserve all of the state pertinent to the continuation of the call. Some aspects of the state will be explicitly available (such as the values of local variables), but other aspects may not be. A prime example is the remainder of the current call stack. For languages that do not provide support for first-class continuations, like Java and C$^\sharp$, accounting for this state often requires a manual transformation to continuation-passing-style of not only the enclosing method, but also all of its callers. Once reified as an explicit continuation, the state of a computation can be saved at the initiation of an asynchronous operation and restored on its completion by supplying it with a result.

The upcoming version of C$^\sharp$ (and Visual Basic) features dedicated linguistic support for asynchronous programming that removes the need for explicit callbacks. C$^\sharp$ 5.0 allows certain methods to pause and then later resume their computation, without blocking, at explicitly marked code points. The basic idea is to allow a method, designated as asynchronous, to *await* the completion of some other event, not by blocking its executing thread, but by *pausing* its own execution and releasing its thread to do further work. The caller of the paused method then receives a *task* representing the method's future result and is free to proceed. Subsequent completion of the awaited event causes the paused method to resume *playing* from where it left off. Since its original thread has carried on, the resumed method is played on some available thread. Depending on run-time context, this thread may be drawn from the .NET thread pool, or it may be the same, issuing thread but at a later opportunity (e.g. the resumed method might be re-enqueued in the user interface's message loop). The events that can be awaited are typically tasks returned by nested calls to asynchronous methods. They can also, more generally, belong to any (user-defined) *awaitable* type or primitive implementations provided by the framework. The aim of these new features is to make it easy to write asynchronous methods, without having to resort to continuation-passing-style and its debilitating inversion of control flow.

As we shall see, the new asychronous features in C$^\sharp$ 5.0 are quite subtle. Current Microsoft specifications describe the features using precise prose and by example, giving illustrative source-to-source translations from C$^\sharp$ 5.0 to ordinary C$^\sharp$ 4.0. Unfortunately, the translation is intricate and its output verbose and difficult to comprehend as it generates internal finite state machines. We believe a formal, mathematical approach can yield both a precise foundation for researchers, but also a better mental model for developers and compiler writers to justify the correctness of their translation. The primary contribution of this paper is to provide such a model: a direct, operational semantics of the feature in a representative fragment of C$^\sharp$ 5.0. Our semantics both capture the intent of the feature and explain its performance-driven limitations, without appealing to low-level compiler output. The techniques we present should scale to provide high-level accounts of a closely-related and similarly documented feature: C$^\sharp$ iterators.

The paper is structured as follows. §2.1 gives an informal yet precise description of the feature. §2.2 presents a realistic example, re-coding a non-trival synchronous method to an asynchronous one, first using the feature, then contrasting it with its equivalent, hand-crafted implementation. §3 formally describes our core fragment of C$^\sharp$ 5.0 and presents both a type system and an operational semantics. §4 sketches some of the correctness properties that our formalization satisfies. §5 presents some extensions to our basic setting; in §5.1 we show how to develop our operational semantics to be less abstract, and much closer to the implementation but still without having to resort to a finite state machine translation. In §5.2 we show how to extend our formalization to capture the awaitable pattern. §6 surveys briefly related work on asynchronous programming; and §7 presents conclusions and some future work.

2 Background

2.1 Async C$^\sharp$ extensions in a nutshell

Syntactically, the additions to C$^\sharp$ are surprisingly few: one new modifier `async` to mark a member as asynchronous and one new *expression*, `await` e, for awaiting the result—control, a value or exception—of some *awaitable* expression. An `await` expression can also be used as a statement, `await` e`;`, discarding its value. The `async` modifier can be placed on methods (excluding iterators) and some other method-like constructs (anonymous, first-class methods, i.e. lambdas and delegates). An `await` expression can only appear in an `async` method; other occurrences are static errors.

Statically, an `async` method must have a *taskable* return type of `Task`<σ>, `Task` or `void`. The `return` *statements* of an `async` method with return type `Task`<σ> may only return values of type σ, never `Task`<σ>! The return statements in other `async` methods may only return control (but never a value).

The argument, e, of an `await` e expression must have an *awaitable* type. The concept of awaitable type is defined by a *pattern* (of available methods). A type is awaitable when it statically supports a `GetAwaiter()` instance method that returns some *awaiter* type (possibly the same type). In turn, the awaiter type must support:

- a boolean instance property `IsCompleted` testing if the awaiter has a result now.
- a `void`-returning instance method `OnCompleted(`a`)`, accepting a callback of delegate type `Action`.[3] Action a is a *one-shot* continuation; calling a`()` resumes the awaiting method. The action should be invoked *at most once*, on completion.
- a τ-returning instance method `GetResult()` for retrieving the result of a completed awaiter. `GetResult()` should either return control, some stored value, or throw some stored exception.

If the return type of `GetResult` is τ, then expression `await` e is an *expression* of type τ. All these operations should be (essentially) *non-blocking*.

Crucially, the types `Task`<σ> and `Task` are awaitable, allowing `async` methods to await the tasks of *nested* `async` method calls. A caller can use a returned task just like any other task (asynchronously awaiting its result, synchronously waiting for its completion or by registering an asynchronous callback). Asynchronous methods that return `void` cannot be awaited; such methods are intended for 'fire-and-forget' scenarios.

Dynamically, an `async` method executes like an ordinary method until it encounters an `await` on some value. If the value's awaiter is complete, the method continues executing using the result of the awaiter as the result of the await expression. If the awaiter is incomplete, the method registers its continuation as a callback on the awaiter and then suspends its execution. Execution will resume with the result of the awaiter, on some thread,[4] when the callback is invoked. Invoking an async method allocates

[3] Defined as `delegate void Action()`.

[4] We are deliberately vague here; the awaiter is free to choose how the method is resumed, catering for different behaviours in, for example, single- and multi-threaded applications.

a fresh, incomplete task, representing this invocation, immediately enters the method (on the caller's thread) and executes it until it encounters its first await on an incomplete awaiter. Exiting from an async method, either via return or throwing an exception, stores the result in its task, thus *completing* it. The first suspension of an async method call returns its incomplete task (or void) to its caller. If the call exits without ever suspending, it simply returns its completed task.

The operational semantics is deliberately designed to minimize context switches. Continuing when an awaiter is already complete ensures methods only suspend when necessary. Dually, allowing an `async` method call to begin execution on its caller's thread gives the method the opportunity to enter-and-exit quickly when possible, without imposing the cost of a context switch just to get running in the first place. This design choice morally obliges the method not to block nor even spend too much time before ceding its caller's thread (by suspension).

2.2 Example

To both illustrate and motivate the feature we present an example adapted from the Async CTP.[5] Consider the following synchronous copy function, which incrementally copies an input stream to an output stream in manageable chunks.

```
public static long CopyTo(Stream src, Stream dst) {
  var buffer = new byte[0x1000];
  int bytesRead;
  long totalRead = 0;
  while ((bytesRead = src.Read(buffer, 0, buffer.Length)) > 0) {
    dst.Write(buffer, 0, bytesRead);
    totalRead += bytesRead;
  }
  return totalRead;
}
```

Depending on their receivers, the calls to `Stream` methods `Read` and `Write` may well be blocking IO operations. Since this method could spend much of its time blocked, one might prefer an asynchronous variant. One way to achieve this is by replacing the synchronous calls to `Read` and `Write` with their asynchronous counterparts:

```
Task<int> ReadAsync(byte[] buffer, int offset, int count);
Task WriteAsync(byte[] buffer, int offset, int count);
```

The asynchronous variants initiate an asynchronous operation and immediately return a task representing its completion. Method `ReadAsync` begins a read and immediately returns a `Task<int>` that, once completed, will record the number of bytes actually read. The method `WriteAsync` begins a write and returns a `Task` that just tracks its completion. Tasks are completed at most once with some result. The result may be a value or some exception.

Asynchronous clients can register zero or more callbacks on a task, to be executed (on some thread) once the task has completed with a result, e.g.:

[5] `http://msdn.microsoft.com/vstudio/async`

```
Task<int> rdTask = src.ReadAsync(buffer, 0, buffer.Length);
rdTask.ContinueWith((Task<int> completedRdTask) => {
  int bytesRead = completedRdTask.Result; /* won't block */
  dst.WriteAsync(buffer, 0, bytesRead);
});
// do some work now
```

Since `ContinueWith` is also non-blocking, the client will quickly proceed with its work. The call to property `completedRdTask.Result` in the callback is guaranteed not to block because its task (an alias of `rdTask`) must, by causality, already be completed.

Synchronous clients can also access a task's `Result` property or just `Wait()` for its completion; these calls will block until or unless the task has completed:

```
Task<int> rdTask = src.ReadAsync(buffer, 0, buffer.Length);
// do some work now
int bytesRead = rdTask.Result; /* may block */
Task wrTask = dst.WriteAsync(buffer, 0, bytesRead);
// do some more work now
wrTask.Wait(); /* may block */
```

Now let us show how C$^\sharp$ 5.0 enables the simple implementation of an asynchronous version of `CopyTo`. First we mark the method as `async`, and then simply await the results of `src.ReadAsync` and `dst.WriteAsync`. Otherwise the code remains the same.

```
public static async Task<long> CopyToAsync(Stream src, Stream dst) {
  var buffer = new byte[0x1000];
  int bytesRead;
  long totalRead = 0;
  while ((bytesRead = await src.ReadAsync(buffer, 0, buffer.Length)) > 0) {
    await dst.WriteAsync(buffer, 0, bytesRead);
    totalRead += bytesRead;
  }
  return totalRead;
}
```

Awaiting the task returned by `ReadAsync` pauses the method unless the read has completed; when play is resumed, the await expression extracts the integer value of the task. `WriteAsync` returns a non-generic `Task`; the await statement pauses the method unless the write has completed; when it is played again, execution proceeds from the next statement. Note that the return type of our asynchronous method is `Task<long>` even though its body `return`s a `long`. Clearly, this is no ordinary `return` statement.

Though the code is almost identical to the original, the behaviour and resource consumption is quite different. A call to `CopyTo` will repeatedly block (in the kernel) on each call to `Read` and `Write`, tying up the resources dedicated to that thread. A call `CopyToAsync`, on the other hand, will never block; instead, each continuation of an await will be executed on demand, on some available thread in the .NET thread pool.[6]

To appreciate the concision of `CopyToAsync`, let us contrast it with a representative hand-crafted version of `CopyToAsync`, `CopyToManual`, written in C$^\sharp$ 4.0. Actually, this

[6] If invoked from a user interface thread, each continuation will be scheduled on that thread's event queue.

code is very close to the decompiled code emitted by the C♯ 5.0 compiler and described in the feature documentation. As mentioned earlier, the aim of our work is to eliminate the need to understand this compilation strategy.

```
public static Task<long> CopyToManual(Stream src, Stream dst) {
  var tcs = new TaskCompletionSource<long>(); // tcs.Task is an incomplete task
  byte[] buffer = null; int bytesRead = 0; long totalRead = 0;
  TaskAwaiter<int> readAwaiter;
  TaskAwaiter writeAwaiter;
  var state = 0;
  Action act = null;
  act = () => {
    while (true) {
      switch (state) {
        case 0: // initial state
          buffer = new byte[0x1000];
          totalRead = 0;
          state = 1;
          break;
        case 1: // loop header
          state = 2;
          readAwaiter = src.ReadAsync(buffer, 0, buffer.Length).GetAwaiter();
          if (readAwaiter.IsCompleted) { }
          else { readAwaiter.OnCompleted(act); return;} // suspend at 2
          break;
        case 2: // continuation of read await
          if ((bytesRead = readAwaiter.GetResult()) > 0) {
            state = 3;
            writeAwaiter = dst.WriteAsync(buffer, 0, bytesRead).GetAwaiter();
            if (writeAwaiter.IsCompleted) { }
            else { writeAwaiter.OnCompleted(act); return;} // suspend at 3
          }
          else state = 4; // goto continuation of while
          break;
        case 3: // continuation of write await
          writeAwaiter.GetResult(); // throw any exception
          totalRead += bytesRead;
          state = 1; // goto loop header
          break;
        case 4: // continuation of while
          tcs.SetResult(totalRead); // complete tcs.Task
          return; // exit machine
  }}};
  act(); // start the machine on this thread
  return tcs.Task; // on first suspend or exit from machine
}
```

For simplicity, we omit the code required to catch and propagate exceptions in the returned task. Without going into too many details, notice how the control flow has been obscured by encoding the continuation of each await as states (here 2 & 3) of a finite state machine. The original locals, arguments and internal state of the method are (implicitly) allocated on the heap. (Note that C♯ lambdas such as `act` close over L-values, not R-values, automatically placing them on the heap; updates to those locations persist across lambda invocations.) State 0 is the initial state that sets up locals; state 1 is the `while` loop header, state 2 and 3 are the continuations of the `await` statements; state 4 is the final state and the continuation of the original `while` statement. State 4 exits the machine, setting the result in the task held by shared variable `tcs`. The finite state machine only suspends (by calling `return` without completing the task) in state 2 and 3 (just after an await); the other states encode internal control flow points.

2.3 Parallelism and Overlapped Task Execution

`CopyToAsync`'s reads and writes are still being executed sequentially, but from different threads rather than a single one, so we would not expect to gain any performance from the asynchronous implementation. Indeed, given the additional scheduling and compilation overheads, the synchronous `CopyTo` is likely to execute faster.

However, we are now in a good position to *overlap* the last write with the next read, leading to this faster, parallel implementation:

```
public static async Task<long> CopyToParallel(Stream src, Stream dst) {
  var buffer = new byte[0x1000]; var oldbuffer = new byte[0x1000];
  int bytesRead;
  long totalRead = 0;
  Task lastwrite = null;
  while ((bytesRead = await src.ReadAsync(buffer,0,buffer.Length)) > 0) {
    if (lastwrite != null) await lastwrite; // ask later
    lastwrite = dst.WriteAsync(buffer, 0, bytesRead); // shoot first
    totalRead += bytesRead;
    { var tmp = buffer; buffer = oldbuffer; oldbuffer = tmp; };
  }
  if (lastwrite != null) await lastwrite;
  return totalRead;
}
```

In order to achieve this, we exploit the ability to separate the initiation of a task from the act of awaiting its completion, so that we can issue the next read *during* the last write. The example illustrates the crucial advantage of allowing asynchronous methods to return incomplete, concurrently executing tasks, not just completed results.

3 Formalization: Featherweight C$^\sharp$ 5.0

In the rest of the paper we study the essence of the new asynchronous features of C$^\sharp$. To do so we take a formal, mathematical approach and define an idealized fragment, Featherweight C$^\sharp$ 5.0, or $\mathsf{FC}^\sharp_5$ for short. Whilst $\mathsf{FC}^\sharp_5$ programs remain syntactically valid C$^\sharp$, it is a heavily restricted fragment—any language feature that is not needed to demonstrate the essence of the asynchronous features has been removed, and the resulting fragment has been further refactored to allow for a more succinct presentation.

As the new asynchronous features essentially interact with the control flow of C$^\sharp$ program, most of our attention is on the operational semantics. In contrast, other minimal fragments such as, for example, Featherweight Java [12], and Classic Java [9], are primarily concerned with typing issues. The asynchronous features in C$^\sharp$ 5.0 have almost no impact on the type system. Consequently we have stripped the type system of $\mathsf{FC}^\sharp_5$ to the core: we have only simple non-generic classes, some value types and no subtyping at all!

$\mathsf{FC}^\sharp_5$ programs and types:

$p ::= \overline{cd}\ mb$	Program
$cd ::=$	Class declaration
`public class` `C` `{`$\overline{fd}\ \overline{md}$`}`	
$fd ::=$ `public` $\sigma\ f$`;`	Field declaration
$md ::=$	Method declaration
`public` $\phi\ m($$\overline{\sigma}\ \overline{x}$$)\ mb$	
`async public` $\psi\ m($$\overline{\sigma}\ \overline{x}$$)\ mb$	
$mb ::=$ `{` $\overline{\sigma\ x}$`;`$\overline{s}$ `}`	Method body
$\phi ::=$	Return type
σ	Type
`void`	Void type
$\sigma, \tau ::=$	Type
γ	Value type
`C`	Class type
	(including a family of types `Task`<σ>)
$\gamma ::=$	Value Type
`bool`	Boolean
`int`	Integer
$\psi ::=$ `Task`<σ>	Taskable return type

Our formalization makes heavy use of the Featherweight Java [12] overbar notation, i.e. we write $\overline{x}$ for a possibly empty sequence $x_1, \ldots x_n$. We abbreviate operations on pairs of sequences, writing for example $\overline{\sigma}\ \overline{x}$ for the sequence $\sigma_1\ x_1, \ldots, \sigma_n\ x_n$, similarly $\overline{\sigma\ x;}$ for the sequence of variable declarations $\sigma_1\ x_1;\ \ldots \sigma_n\ x_n;$ and finally $f(\overline{\sigma})$ for the sequence $f(\sigma_1), \ldots, f(\sigma_n)$.

A $\mathsf{FC}^\sharp_5$ program consists of a collection of class declarations and a single method body (C$^\sharp$'s `main` method). [7]A $\mathsf{FC}^\sharp_5$ class declaration `public class` `C` `{`$\overline{fd}\ \overline{md}$`}` introduces a class `C`. We repeat that $\mathsf{FC}^\sharp_5$ does not support any form of subtyping so class declarations do not specify a superclass. This is a valid declaration in full C$^\sharp$ as all classes inherit from `object` by default, but we do not even support the `object` class in $\mathsf{FC}^\sharp_5$. Subtyping and inheritance are orthogonal to the new features in C$^\sharp$ 5.0 and so we removed them from our fragment to concentrate solely on the support for asynchronous programming.[8] The class `C` has fields $\overline{f}$ with types $\overline{\sigma}$ and a collection of methods $\overline{md}$. All members are required to be public.

Method declarations can be either synchronous or asynchronous. A synchronous method `public` $\phi\ m($$\overline{\sigma}\ \overline{x}$$)\ mb$ declares a public method m with return type ϕ, formal parameters $\overline{x}$ of type $\overline{\sigma}$ and a method body mb. Methods may be `void`-returning, i.e.

[7] This is a modest extension to C$^\sharp$ that restricts the `main` method to being `void` or `int` returning.

[8] The extensions to support single inheritance, overloading, constructor methods and many of the complications of the full C$^\sharp$ type system have appeared in other (synchronous) featherweight fragments [2, 3].

they return control not a value. Method bodies are constrained to be of a particular form: $\overline{\sigma}\ \overline{x}$;$\overline{s}$, i.e. they must declare all their local variables $\overline{x}$ at the start of the method, and then contain a sequence of statements $\overline{s}$.

An asynchronous method is marked with the `async` keyword and is syntactically the same as a synchronous method, although it is type checked differently. The return type of an asynchronous method must be of a so-called taskable type. For $\mathsf{FC}_5^\sharp$ this means it must be of the form `Task`<σ>. C$^\sharp$ 5.0 also classifies the non-generic class `Task` and `void` as taskable return types as discussed in §2.1.

$\mathsf{FC}_5^\sharp$ types are a simple subset of the C$^\sharp$ types. Note that $\mathsf{FC}_5^\sharp$ does not support user-defined generics; again these are orthogonal to asynchrony and have been removed. For simplicity, we assume that `Task`<σ> is the only generic type.

$\mathsf{FC}_5^\sharp$ expressions and statements:

e ::=	Expressions
$\underline{b}$	Boolean
$\underline{i}$	Integer
$x \oplus y$	Built-in operator
x	Variable
`null`	Null
x.f	Field access
x.m($\overline{y}$)	Method invocation
`new` C()	Object creation
`await` x	Await expression
`Task`.AsyncIO<γ>()	Async primitive
s ::=	Statement
x=e;	Assignment statement
`if` (x) {$\overline{s}$} `else` {$\overline{t}$}	Conditional statement
`while` (x) {$\overline{s}$}	Iteration
x.f = y;	Field assignment statement
x.m($\overline{y}$);	Method invocation statement
`return`;	Return statement
`return` x;	Return value statement

$\mathsf{FC}_5^\sharp$ expressions are restricted to a form that we call *statement normal form (SNF)*. SNF forces all subexpressions to be named; i.e. all subexpressions are simply variables. SNF is the natural analogue to the A-normal-form popular in functional languages [8]. This regularity makes the presentation of the operational semantics (and the type system) much simpler at no cost to expressivity.

$\mathsf{FC}_5^\sharp$ expressions include constants, which can be an integer, i, a boolean, b, or the literal `null`. We assume a number of built-in primitive operators, such as ==, <, > and so on. In the grammar we write $x \oplus y$, where $\oplus$ denotes an instance of one of these operators. We do not specify operators further as their meaning is clear. We assume that x, y, z range over variable names, f ranges over field names and m ranges over method names. We assume that the set of variables includes the special variable `this`, which cannot be used as a parameter of a method declaration or declared as a local.

$\mathsf{FC}_5^\sharp$ supports awaitable expressions, written `await` x. To get things off the ground we assume an in-built asynchronous method `Task`.AsyncIO<γ>() that spawns a thread and immediately returns a task. The thread may complete the task, depending on the scheduler, with some result of value type γ.

$\mathsf{FC}^\sharp_5$ statements are standard. In what follows we assume that $\mathsf{FC}^\sharp_5$ programs are well-formed, e.g. the identifier `this` does not appear as a formal parameter, all control paths in a method body contain a `return` statement, etc. These conditions can be easily formalized and are identical to restrictions on earlier fragments of C$^\sharp$ but we suppress the details for lack of space. The only new well-formedness condition is that `await` expressions are only allowed to appear inside *asynchronous* method declarations.

We assume that a correct program induces a number of utility functions that we will use in the typing rules. First, we assume the partial function *ftype*, which is a map from a type and a field name to a type. Thus $\mathit{ftype}(\sigma, f)$ returns the type of field f in type σ. Second, we assume a partial function mtype that is a map from a type and a method name to a *type signature*. For example, we write $\mathit{mtype}(\mathtt{C}, m) = (\overline{\tau}) \to \phi$ when class `C` contains a method m with formal parameters of type $\overline{\tau}$ and return type ϕ.

The type system for full C$^\sharp$ is actually a bidirectional type system [16] consisting of two typing relations: a type conversion relation and a type synthesis relation [3], along with a number of conversion (subtyping) judgements. However, the extreme parsimony of $\mathsf{FC}^\sharp_5$ means that we have no subtyping judgements, and we need only a single judgement for type checking an expression. The judgement is written $\Gamma \vdash e\colon \sigma$ where Γ is a function from variables to types. We extend the overbar notation and write $\Gamma \vdash \overline{e}\colon \overline{\sigma}$ to mean the judgements $\Gamma \vdash e_1\colon \sigma_1, \ldots, \Gamma \vdash e_n\colon \sigma_n$.

$\mathsf{FC}^\sharp_5$ expression type checking:

$$\Gamma \vdash \underline{b}\colon \mathtt{bool} \qquad \text{[C-Bool]}$$

$$\Gamma \vdash \underline{i}\colon \mathtt{int} \qquad \text{[C-Int]}$$

$$\frac{\Gamma \vdash x\colon \sigma_0 \qquad \Gamma \vdash y\colon \sigma_1 \qquad \oplus\colon \sigma_0 \times \sigma_1 \to \tau}{\Gamma \vdash x \oplus y\colon \tau} \qquad \text{[C-Op]}$$

$$\Gamma, x\colon \tau \vdash x\colon \tau \qquad \text{[C-Var]}$$

$$\frac{\mathit{ftype}(\sigma, f) = \tau}{\Gamma, x\colon \sigma \vdash x\,.\,f\colon \tau} \qquad \text{[C-Field]}$$

$$\Gamma \vdash \mathtt{new}\ \mathtt{C}()\colon \mathtt{C} \qquad \text{[C-New]}$$

$$\frac{\mathit{mtype}(\sigma_0, m) = (\overline{\tau}) \to \sigma_1 \qquad \Gamma, x\colon \sigma_0 \vdash \overline{y}\colon \overline{\tau}}{\Gamma, x\colon \sigma_0 \vdash x\,.\,m\,(\overline{y})\colon \sigma_1} \qquad \text{[C-MethInv]}$$

$$\Gamma, x\colon \mathtt{Task}\texttt{<}\sigma\texttt{>} \vdash \mathtt{await}\ x\colon \sigma \qquad \text{[C-Await]}$$

$$\Gamma \vdash \mathtt{Task.AsyncIO}\texttt{<}\gamma\texttt{>}()\colon \mathtt{Task}\texttt{<}\gamma\texttt{>} \qquad \text{[C-IO]}$$

Most of the type checking rules are quite standard, but there are two new rules for dealing with asynchronous methods. Rule [C-Await] states that if x is of type `Task`<σ> then awaiting x results in a value of type σ. As discussed earlier, Rule [C-IO] states that `Task.AsyncIO`<γ>`()` returns a value of type `Task`<γ>.

$\mathsf{FC}^\sharp_5$ statement type checking:

$$\frac{\Gamma, x\colon \sigma \vdash e\colon \sigma}{\Gamma, x\colon \sigma \vdash x \texttt{ = } e\texttt{;}\colon \phi} \qquad \text{[C-Asn]}$$

$$\frac{\Gamma, x\colon \texttt{bool} \vdash \overline{s}\colon \phi \qquad \Gamma, x\colon \texttt{bool} \vdash \overline{t}\colon \phi}{\Gamma, x\colon \texttt{bool} \vdash \texttt{if}\ (x)\ \{\overline{s}\}\ \texttt{else}\ \{\overline{t}\}\colon \phi} \quad \text{[C-Cond]}$$

$$\frac{\Gamma, x\colon \texttt{bool} \vdash \overline{s}\colon \phi}{\Gamma, x\colon \texttt{bool} \vdash \texttt{while}\ (x)\ \{\overline{s}\}\colon \phi} \quad \text{[C-While]}$$

$$\frac{\mathit{ftype}(\sigma_0, f) = \sigma_1 \qquad \Gamma, x\colon \sigma_0 \vdash y\colon \sigma_1}{\Gamma, x\colon \sigma_0 \vdash x\,.\,f\texttt{=}y\texttt{;}\colon \phi} \quad \text{[C-FAsn]}$$

$$\frac{\mathit{mtype}(\sigma_0, m) = (\overline{\tau}) \to \texttt{void} \qquad \Gamma, x\colon \sigma_0 \vdash \overline{y}\colon \overline{\tau}}{\Gamma, x\colon \sigma_0 \vdash x\,.\,m\texttt{(}\overline{y}\texttt{)}\texttt{;}\colon \phi} \quad \text{[C-MInv]}$$

$$\Gamma \vdash \texttt{return;}\colon \texttt{void} \quad \text{[C-Return]}$$

$$\Gamma, x\colon \sigma \vdash \texttt{return}\ x\texttt{;}\colon \sigma \quad \text{[C-ReturnExp]}$$

As for full C♯, we give type checking rules for statements; the judgement is written $\Gamma \vdash s\colon \phi$. The key rules are [C-ReturnExp] that asserts that the statement `return` x`;` is of return type σ if x is of type σ and [C-Return] that asserts that the statement `return;` is of return type `void`. In other words, the role of the type ϕ in the judgement $\Gamma \vdash s\colon \phi$ is to check any return statement. Again, we adopt an overbar notation and write $\Gamma \vdash \overline{s}\colon \phi$ to denote the judgements $\Gamma \vdash s_1\colon \phi, \ldots, \Gamma \vdash s_n\colon \phi$.

$\mathsf{FC}_5^\sharp$ method and class typing:

$$\frac{\overline{x}\colon \overline{\sigma}, \overline{y}\colon \overline{\tau}, \texttt{this}\colon \texttt{C} \vdash \overline{s}\colon \phi}{\texttt{C} \vdash \texttt{public}\ \phi\ m\texttt{(}\overline{\sigma}\ \overline{x}\texttt{)}\texttt{\{}\overline{\tau}\ \overline{y}\texttt{;}\overline{s}\texttt{\}}\ \mathbf{ok}} \quad \text{[Meth-OK]}$$

$$\frac{\overline{x}\colon \overline{\sigma}, \overline{y}\colon \overline{\tau}, \texttt{this}\colon \texttt{C} \vdash \overline{s}\colon \sigma_0}{\texttt{C} \vdash \texttt{async public Task<}\sigma_0\texttt{>}\ m\texttt{(}\overline{\sigma}\ \overline{x}\texttt{)}\texttt{\{}\overline{\tau}\ \overline{y}\texttt{;}\overline{s}\texttt{\}}\ \mathbf{ok}} \quad \text{[AsyncMeth-OK]}$$

$$\frac{\texttt{C} \vdash \overline{\mathit{md}}\ \mathbf{ok}}{\vdash \texttt{public class C \{}\overline{\mathit{fd}}\ \overline{\mathit{md}}\ \texttt{\}}\ \mathbf{ok}} \quad \text{[Class-OK]}$$

Rule [Class-OK] asserts that a class declaration is well-typed provided that all the method declarations are well-typed. Rule [Meth-OK] asserts that the (synchronous) method declaration `public` $\phi\ m$`(`$\overline{\sigma}\ \overline{x}$`){`$\overline{\tau}\ \overline{y}$`;`$\overline{s}$`}` is well-typed in class `C` provided that the statements $\overline{s}$ can be typed at return type ϕ in the context $\overline{x}\colon \overline{\sigma}, \overline{y}\colon \overline{\tau}$, `this`: `C`. Moreover, $\overline{s}$ cannot contain `await`. Rule [AsyncMeth-OK] asserts that the asynchronous method `async public Task<`σ_0`>` m`(`$\overline{\sigma}\ \overline{x}$`){`$\overline{\tau}\ \overline{y}$`;`$\overline{s}$`}` is well-typed if the statements $\overline{s}$ can be typed at return type σ_0 (not `Task<`σ_0`>`) in the context $\overline{x}\colon \overline{\sigma}, \overline{y}\colon \overline{\tau}$, `this`: `C`.

We assume two methods on the `Task<`σ`>` type, `Result` and `GetResult`, which both take no argument and return a value of type σ, i.e. $\mathit{mtype}(\texttt{Task<}\sigma\texttt{>}, \_) = () \to \sigma$.[9] Both these methods return the result of a complete task object, but will differ operationally on an incomplete task.

3.1 Operational semantics

The key contribution of this paper is a precise description of the operational behaviour of the new asynchronous features in C♯. The syntactic restrictions of $\mathsf{FC}_5^\sharp$ mean that the operational semantics can be given quite straightforwardly as single-step transition rules between configurations.

[9] In C♯, `Result` is actually a property rather than a method. The difference is cosmetic.

A *heap*, H, is a partial map from an object identifier (ranged over by o) to a heap object. A *heap object*, $\langle \mathtt{C}, FM \rangle$, is a pair of a type $\mathtt{C}$ and a *field map*, FM, which is a partial map from fields f to values (ranged over by v). A *value* is either an integer, a boolean, the `null` literal or an object identifier (the *address* of an object in the heap).

A *frame*, F, is written $\langle L, \overline{s} \rangle^{\ell}$ and consists of a locals stack, L, and a sequence of statements, $\overline{s}$, along with a frame label, ℓ. A *locals stack* is a partial map from local variables to values. A *frame label*, ℓ, is either $\mathtt{s}$ to denote a synchronous frame, or $\mathtt{a}(o)$ for an asynchronous frame whose associated `Task` is stored at heap address o. A *frame stack*, FS, is essentially a list of frames. An empty frame stack is written ϵ, and we write $F \circ FS$ to denote a frame stack whose head is a frame F and tail is the frame stack FS. A process, P, is a collection of frame stacks, written $\{FS_1, \ldots, FS_n\}$.

We factor the transition rules into three relations describing the small step evaluation of frames (method bodies), frame stacks (corresponding to individual threads) and collections of frame stacks (corresponding to a process, i.e. a pool of threads mutating a shared heap). Thus, a frame configuration is written $H \rhd F$ and the transition relation between frame configurations is written $H_1 \rhd F_1 \rightarrow H_2 \rhd F_2$. A frame stack configuration is written $H \rhd FS$ and the transition relation between frame stack configurations is written $H_1 \rhd FS_1 \twoheadrightarrow H_2 \rhd FS_2$. Finally, a process configuration is written $H \rhd P$ and the transition relation between process configurations is written $H_1 \rhd P_1 \rightsquigarrow H_2 \rhd P_2$.

Simple frame transition rules:

$H \rhd \langle L, x\mathtt{=}v\mathtt{;}\overline{s} \rangle^{\ell} \rightarrow H \rhd \langle L[x \mapsto v], \overline{s} \rangle^{\ell}$ [E-Value]

$H \rhd \langle L, x\mathtt{=}y\mathtt{;}\overline{s} \rangle^{\ell} \rightarrow H \rhd \langle L[x \mapsto L(y)], \overline{s} \rangle^{\ell}$ [E-Var]

$H \rhd \langle L, x\mathtt{=}y \oplus z\mathtt{;}\overline{s} \rangle^{\ell} \rightarrow H \rhd \langle L[x \mapsto L(y) \oplus L(z)], \overline{s} \rangle^{\ell}$ [E-Op]

$H \rhd \langle L, x\mathtt{=}y\mathtt{.}f\mathtt{;}\overline{s} \rangle^{\ell}$ [E-Field]
$\rightarrow H \rhd \langle L[x \mapsto FM(f)], \overline{s} \rangle^{\ell}$ where $L(y) = o$ and $H(o) = \langle \sigma, FM \rangle$

$H \rhd \langle L, \mathtt{if}\ \mathtt{(}x\mathtt{)}\ \mathtt{\{}\overline{s}\mathtt{\}}\ \mathtt{else}\ \mathtt{\{}\overline{t}\mathtt{\}}\ \overline{u} \rangle^{\ell}$ [E-CondEq]
$\rightarrow H \rhd \langle L, \overline{s}\ \overline{u} \rangle^{\ell}$ where $L(x) = \mathtt{true}$
$\rightarrow H \rhd \langle L, \overline{t}\ \overline{u} \rangle^{\ell}$ where $L(x) = \mathtt{false}$
$H \rhd \langle L, \mathtt{while}\ \mathtt{(}x\mathtt{)}\mathtt{\{}\overline{s}\mathtt{\}}\ \overline{t} \rangle^{\ell}$ [E-While]
$\rightarrow H \rhd \langle L, \overline{s}\ \mathtt{while}\ \mathtt{(}x\mathtt{)}\mathtt{\{}\overline{s}\mathtt{\}}\ \overline{t} \rangle^{\ell}$ where $L(x) = \mathtt{true}$
$\rightarrow H \rhd \langle L, \overline{t} \rangle^{\ell}$ where $L(x) = \mathtt{false}$

$H_0 \rhd \langle L, x\mathtt{.}f\mathtt{=}y\mathtt{;}\overline{s} \rangle^{\ell} \rightarrow H_1 \rhd \langle L, \overline{s} \rangle^{\ell}$ [E-Asn]
where $L(x) = o, H_0(o) = \langle \sigma, FM \rangle$ and $H_1 = H_0[o \mapsto \langle \sigma, FM[f \mapsto L(y)] \rangle]$

$H_0 \rhd \langle L, x\mathtt{=new}\ \mathtt{C()}\mathtt{;}\overline{s} \rangle^{\ell} \rightarrow H_1 \rhd \langle L[x \mapsto o], \overline{s} \rangle^{\ell}$ [E-New]
where $\mathit{fields}(\mathtt{C}) = \overline{\tau}\ \overline{f}, o \notin \mathit{dom}(H_0)$ and $H_1 = H_0[o \mapsto \langle \mathtt{C}, \overline{f} \mapsto \mathit{default}(\overline{\tau}) \rangle]$

In these transition rules the frames are labeled with meta-variable ℓ: they apply for both synchronous and asynchronous frames, factoring common semantics. Our transition rules $H_0 \rhd \langle L_0, \overline{s} \rangle^{\ell} \rightarrow H_1 \rhd \langle L_1, \overline{t} \rangle^{\ell}$ always preserve labels, i.e. a synchronous frame transitions to another synchronous frame, and an asynchronous frame transitions to an asynchronous frame with the *same* task. In rule [E-New] we use an auxiliary function, *default* that returns a default value for a given type. This notion is taken from full C$^\sharp$ [11, §5.2] but for FC$^\sharp_5$ it simply maps type `int` to the value 0, type `bool` to the value `false` and all other types to the `null` literal. These simple transition rules are quite standard and for space reasons we do not elaborate on them further.

Next we consider the evaluation of a synchronous method call and returning from a synchronous method. Clearly here the label on the frame is important; these rules do not apply for asynchronous methods where evaluation is different.

Synchronous method call/return transition rules:

$H_1 \triangleright F_1 \circ FS \twoheadrightarrow H_2 \triangleright F_2 \circ FS \qquad \text{if } H_1 \triangleright F_1 \rightarrow H_2 \triangleright F_2$ [E-Frame]

$H \triangleright \langle L_0, y_0\texttt{=}y_1\texttt{.}m\texttt{(}\overline{z}\texttt{)}\texttt{;}\overline{s}\rangle^\ell \circ FS \twoheadrightarrow H \triangleright \langle L_1, \overline{t}\rangle^{\mathsf{s}} \circ \langle L_0, \overline{s}\rangle^\ell_{y_0} \circ FS$ [E-Method-Exp]
where $H(L_0(y_1)) = \langle \sigma_0, FM\rangle$
$mbody(\sigma_0, m) = mb\colon (\overline{\sigma}\ \overline{x}) \rightarrow^{\mathsf{s}} \sigma_1$
$mb = \overline{\tau}\ \overline{y}\texttt{;}\ \overline{t}$ and $L_1 = [\overline{x} \mapsto L_0(\overline{z}), \overline{y} \mapsto default(\overline{\tau}), \texttt{this} \mapsto L_0(y_1)]$

$H \triangleright \langle L_0, x\texttt{.}m\texttt{(}\overline{y}\texttt{)}\texttt{;}\overline{s}\rangle^\ell \circ FS \twoheadrightarrow H \triangleright \langle L_1, \overline{t}\rangle^{\mathsf{s}} \circ \langle L_0, \overline{s}\rangle^\ell \circ FS$ [E-Method-Stmt]
where $H(L_0(x)) = \langle \sigma_1, FM\rangle$
$mbody(\sigma_1, m) = mb\colon (\overline{\sigma}\ \overline{x}) \rightarrow^{\mathsf{s}} \texttt{void}$
$mb = \overline{\tau}\ \overline{z}\texttt{;}\ \overline{t}$ and $L_1 = [\overline{x} \mapsto L_0(\overline{z}), \overline{z} \mapsto default(\overline{\tau}), \texttt{this} \mapsto L_0(x)]$

$H \triangleright \langle L_0, \texttt{return}\ y\texttt{;}\overline{s}\rangle^{\mathsf{s}} \circ \langle L_1, \overline{t}\rangle^\ell_x \circ FS \twoheadrightarrow H \triangleright \langle L_1[x \mapsto L_0(y)], \overline{t}\rangle^\ell \circ FS$ [E-Return-Val]

$H \triangleright \langle L_0, \texttt{return;}\overline{s}\rangle^{\mathsf{s}} \circ \langle L_1, \overline{t}\rangle^\ell \circ FS \twoheadrightarrow H \triangleright \langle L_1, \overline{t}\rangle^\ell \circ FS$ [E-Return]

These transition rules are also quite standard. Rule [E-Frame] transitions the topmost, *active* frame of a frame stack. Rule [E-Method-Exp] transitions a method invocation. It first looks up in the heap the runtime type of the receiver. We make use of another auxiliary function induced by correct program: *mbody* is a map from a type and a method name to a method body and an annotated type signature. For example, we write $mbody(\texttt{C}, m) = mb\colon (\overline{\sigma}\ \overline{x}) \rightarrow^{\mathsf{s}} \phi$, when the method m in class `C` is a synchronous method, with formal parameters $\overline{\sigma}\ \overline{x}$, return type ϕ, and method body mb.

Rule [E-Method-Exp] applies when the receiver object supports method m and m is a synchronous method. In this case, we push a new synchronous frame (labeled s) on to the frame stack to execute the method body. Notice that we annotate the caller frame with the identifier that is waiting for the return value (this will be used in rule [E-Return-Val]). Rule [E-Method-Stmt] is similar except that m is a `void`-returning method, returning control.

Rule [E-Return-Val] shows how a synchronous method returns a value to its caller. The caller frame, $\langle L_1, \overline{t}\rangle^\ell_x$, is waiting for a value for local identifier x. The active synchronous frame is popped and the caller frame becomes active and assigns the return value to x. Rule [E-Return] is similar except that no value is returned and the caller frame is not annotated with an identifier: the caller only expects control, not a value.

Asynchronous method call/return transition rules:

$$H_0 \rhd \langle L_0, y_0\texttt{=}y_1\texttt{.}m\texttt{(}\overline{z}\texttt{)}\texttt{;}\overline{s}\rangle^{\ell} \circ FS \qquad \text{[E-Async-Method]}$$
$$\twoheadrightarrow H_1 \rhd \langle L_1, \overline{t}\rangle^{\mathsf{a}(o)} \circ \langle L_0[y_0 \mapsto o], \overline{s}\rangle^{\ell} \circ FS$$
where $H_0(L_0(y_1)) = \langle \sigma_1, FM\rangle$

$mbody(\sigma_1, m) = mb\colon (\overline{\sigma}\ \overline{x}) \rightarrow^{\mathsf{a}} \psi$, and $mb = \overline{\tau}\ \overline{y};\ \overline{t}$

$o \notin dom(H_0), H_1 = H_0[o \mapsto \langle \psi, \texttt{state} \mapsto \mathsf{running}([\,])\rangle]$

$L_1 = [\overline{x} \mapsto L_0(\overline{z}), \overline{y} \mapsto default(\overline{\tau}), \texttt{this} \mapsto L_0(y_1)]$

$$H_0 \rhd \{\langle L, \texttt{return}\ y\texttt{;}\overline{s}\rangle^{\mathsf{a}(o)} \circ FS\} \cup P \qquad \text{[E-Async-Return]}$$
$$\rightsquigarrow H_1 \rhd \{FS\} \cup resume(\overline{F}) \cup P$$
where $H_0(o) = \langle \texttt{Task<}\sigma\texttt{>}, \texttt{state} \mapsto \mathsf{running}(\overline{F})\rangle$

$H_1 = H_0[o \mapsto \langle \texttt{Task<}\sigma\texttt{>}, \texttt{state} \mapsto \mathsf{done}(L(y))\rangle]$

$$H \rhd \langle L, x\texttt{=await}\ y\texttt{;}\overline{s}\rangle^{\mathsf{a}(o)} \circ FS \twoheadrightarrow H \rhd \langle L[x \mapsto v], \overline{s}\rangle^{\mathsf{a}(o)} \circ FS \qquad \text{[E-Await-Continue]}$$
where $H(L(y)) = \langle \texttt{Task<}\sigma\texttt{>}, \texttt{state} \mapsto \mathsf{done}(v)\rangle$

$$H_0 \rhd \langle L, x\texttt{=await}\ y\texttt{;}\overline{s}\rangle^{\mathsf{a}(o)} \circ FS \twoheadrightarrow H_1 \rhd FS \qquad \text{[E-Await]}$$
where $L(y) = o_1, H_0(o_1) = \langle \texttt{Task<}\sigma\texttt{>}, \texttt{state} \mapsto \mathsf{running}(\overline{F})\rangle$

$H_1 = H_0[o_1 \mapsto \langle \texttt{Task<}\sigma\texttt{>}, \texttt{state} \mapsto \mathsf{running}(\langle L, x\texttt{=}y\texttt{.GetResult();}\overline{s}\rangle^{\mathsf{a}(o)}, \overline{F})\rangle]$

$$H \rhd \langle L, x\texttt{=}y\texttt{.Result();}\overline{s}\rangle^{\ell} \rightarrow H \rhd \langle L[x \mapsto v], \overline{s}\rangle^{\ell} \qquad \text{[E-Result]}$$
where $H(L(y)) = \langle \texttt{Task<}\sigma\texttt{>}, \texttt{state} \mapsto \mathsf{done}(v)\rangle$

$$H \rhd \langle L, x\texttt{=}y\texttt{.Result();}\overline{s}\rangle^{\ell} \rightarrow H \rhd \langle L, x\texttt{=}y\texttt{.Result();}\overline{s}\rangle^{\ell} \qquad \text{[E-Result-Block]}$$
where $H(L(y)) = \langle \texttt{Task<}\sigma\texttt{>}, \texttt{state} \mapsto \mathsf{running}(\overline{F})\rangle$

$$H \rhd \langle L, x\texttt{=}y\texttt{.GetResult();}\overline{s}\rangle^{\ell} \rightarrow H \rhd \langle L[x \mapsto v], \overline{s}\rangle^{\ell} \qquad \text{[E-GetResult]}$$
where $H(L(y)) = \langle \texttt{Task<}\sigma\texttt{>}, \texttt{state} \mapsto \mathsf{done}(v)\rangle$

$$H_0 \rhd \{\langle L, x\texttt{=Task.AsyncIO<}\gamma\texttt{>();}\overline{s}\rangle^{\ell} \circ FS\} \cup P \qquad \text{[E-Async-IO]}$$
$$\rightsquigarrow H_1 \rhd \{\langle L[x \mapsto o], \overline{s}\rangle^{\ell} \circ FS\} \cup P \cup \{\langle \{y \mapsto v\}, \texttt{return}\ y\texttt{;}\rangle^{\mathsf{a}(o)} \circ \epsilon\}$$
where $o \notin dom(H_0), H_1 = H_0[o \mapsto \langle \texttt{Task<}\gamma\texttt{>}, \texttt{state} \mapsto \mathsf{running}([\,])\rangle]$ and $v \in Values(\gamma)$

These transition rules cover the new asynchronous features in C♯ 5.0. First we introduce some shorthand for field maps of `Task` objects, where we map a single field `state` to a value $\mathsf{done}(v)$, or $\mathsf{running}(\overline{F})$ where $\overline{F}$ is a list of frames.This use of tagged values is simply a presentational device, in reality we'd use several fields and tags, but this clutters the rules. Tasks are *stateful*: a task object $\langle \texttt{Task<}\sigma\texttt{>}, \texttt{state} \mapsto \_\rangle$ is created in initial state $\mathsf{running}([])$, with no waiters; can transition from state $\mathsf{running}(\overline{F})$ to $\mathsf{running}(F_o, \overline{F})$, adding one waiter, and may terminate in a *completed* state $\mathsf{done}(v)$ for some value v of type σ. Once completed, a task cannot change state again.

Rule [E-Async-Method] shows how to transition a call to an asynchronous method. We create a fresh `Task` object in the heap (at address o), and set its `state` to be running. However, there are no waiters for this task, so the running list is empty (which we write $[\,]$). We push a new frame containing the method body on the frame stack and label it as asynchronous, i.e. with the label $\mathsf{a}(o)$. The caller frame is updated with the heap address of the task in its locals stack. Notice also that the caller frame is *not* waiting for a value, it is simply waiting for control.

[E-Async-Return] pops the active asynchronous frame, storing the return value in the task. It also resumes any waiters (there may be zero or more). The operation $resume(\overline{F})$ is used to resume a collection of suspended frames. It creates a bag of singleton frame stacks and is defined as $resume(\overline{F}) \stackrel{\text{def}}{=} \{\langle L, \overline{s}\rangle^{\ell} \circ \epsilon \mid \langle L, \overline{s}\rangle^{\ell} \in \overline{F}\}$.

Rule [E-Await-Continue] covers the case when a task being awaited is already completed. In this case we simply read out the value from the task and continue. Rule [E-Await] covers the case when the task being awaited is still running. In this case we need to pause the asynchronous method. Thus we pop the active asynchronous frame from the frame stack and add it to the list of awaiters of the incomplete task. Notice that we also insert a new statement into this frame so that when it is resumed the first thing it will do is read the value from the task object in the shared heap.

Rule [E-Result] and [E-Result-Block] implement the in-built method `Result` on tasks. If the task is completed then it returns the result; if it is running then it 'blocks' (which for simplicity we simulate by spinning, i.e. by transitioning to itself). In contrast, rule [E-GetResult] implements `GetResult`. It too returns the result if the task is completed. However, if the task is incomplete, no rule applies and the configuration is stuck (the implementation raises an exception). `GetResult` is *non-blocking* and *partial*.

Rule [E-Async-IO] models a prototypical asynchronous method. Thus a task is created and the frame can continue. A small thread is created that when executed will complete the task with value v.

Process transition rules:

$$H \triangleright \{\epsilon\} \cup P \rightsquigarrow H \triangleright P \qquad \text{[E-Exit]}$$

$$H_0 \triangleright \{FS_0\} \cup P \rightsquigarrow H_1 \triangleright \{FS_1\} \cup P \qquad \text{if } H_0 \triangleright FS_0 \twoheadrightarrow H_1 \triangleright FS_1 \ \text{[E-Schedule]}$$

Recall that a process is a collection of frame stacks, i.e. threads. Rule [E-Exit] deletes an empty frame stack from the process. Rather than formalizing a particular scheduler, rule [E-Schedule] simply transitions a process by non-deterministically selecting and transitioning a thread, possibly side-effecting the shared heap. In other words we have defined an interleaved semantics.

4 Correctness properties

In this section we state some correctness properties that can be proved about $\mathsf{FC}_5^{\sharp}$. First, as is usual, we need to extend the notion of type checking to configurations.

Configuration type checking:

$$\text{[Proc-ok]}\ \frac{\vdash (H \rhd FS_0)\colon \phi_{0,0} \to \phi_{0,1} \quad \cdots \quad \vdash (H \rhd FS_n)\colon \phi_{n,0} \to \phi_{n,1} \qquad \mathit{taskIds}(FS_0) \cap \cdots \cap \mathit{taskIds}(FS_n) = \emptyset}{\vdash (H \rhd \{FS_0, \ldots, FS_n\})\ \mathbf{ok}}$$

$$\text{[T-Config]}\ \frac{\vdash H\ \mathbf{ok} \qquad H \vdash FS\colon \phi_0 \to \phi_1}{\vdash (H \rhd FS)\colon \phi_0 \to \phi_1}$$

$$\text{[H-ok]}\ \frac{\forall o \in \mathit{dom}(H).H(o) = \langle \mathtt{C}, \mathit{FM}\rangle \ \ H \vdash \mathit{FM}\colon \mathtt{C}}{\vdash H\ \mathbf{ok}}$$

$$\text{[T-FM]}\ \frac{\mathit{fields}(\mathtt{C}) = \{\overline{\tau}\ \overline{f}\} \qquad \mathit{FM} = \overline{f} \mapsto \overline{v} \qquad H \vdash \overline{v}\colon \overline{\tau}}{H \vdash \mathit{FM}\colon \mathtt{C}}$$

$$\text{[T-Oid]}\ \frac{H(o) = \langle \mathtt{C}, \mathit{FM}\rangle}{H \vdash o\colon \mathtt{C}}$$

$$\text{[T-Null]}\ \frac{}{H \vdash \mathtt{null}\colon \mathtt{C}} \quad \text{[T-int]}\ \frac{}{H \vdash i\colon \mathtt{int}} \quad \text{[T-bool]}\ \frac{}{H \vdash b\colon \mathtt{bool}}$$

$$\text{[T-EmpFS]}\ \frac{}{H \vdash \epsilon\colon \phi \to \phi} \quad \text{[T-SFS]}\ \frac{\Gamma; H \vdash F^{\mathsf{s}}\colon \phi_0 \to \phi_1 \qquad H \vdash FS\colon \phi_1 \to \phi_2}{H \vdash F^{\mathsf{s}} \circ FS\colon \phi_0 \to \phi_2}$$

$$\text{[T-FS]}\ \frac{\Gamma; H \vdash F^{\mathsf{a}(o)}\colon \phi_0 \to \phi_1 \qquad \mathit{Running}(H(o)) \qquad o \notin \mathit{taskIds}(FS) \qquad H \vdash FS\colon \phi_1 \to \phi_2}{H \vdash F^{\mathsf{a}(o)} \circ FS\colon \phi_0 \to \phi_2}$$

$$\text{[T-SFrame]}\ \frac{\Gamma; H \vdash L\ \mathbf{ok} \qquad \Gamma \vdash \overline{s}\colon \phi}{\Gamma; H \vdash \langle L, \overline{s}\rangle^{\mathsf{s}}\colon \mathtt{void} \to \phi}$$

$$\text{[L-ok]}\ \frac{\mathit{dom}(L) = \mathit{dom}(\Gamma) \qquad \forall x \in \mathit{dom}(L).\ H \vdash L(x)\colon \Gamma(x)}{\Gamma; H \vdash L\ \mathbf{ok}}$$

$$\text{[T-AFrame]}\ \frac{H(o) = \langle \mathtt{Task}\texttt{<}\sigma\texttt{>}, \mathit{FM}\rangle \qquad \Gamma; H \vdash L\ \mathbf{ok} \qquad \Gamma \vdash \overline{s}\colon \sigma}{\Gamma; H \vdash \langle L, \overline{s}\rangle^{\mathsf{a}(o)}\colon \mathtt{void} \to \mathtt{void}}$$

$$\text{[T-NCFrame]}\ \frac{\Gamma, x\colon \sigma; H \vdash L\ \mathbf{ok} \qquad \Gamma, x\colon \sigma; H \vdash \langle L, \overline{s}\rangle^{\ell}\colon \mathtt{void} \to \phi}{\Gamma, x\colon \sigma; H \vdash \langle L, \overline{s}\rangle^{\ell}_{x}\colon \sigma \to \phi}$$

These rules are similar to ones for other configuration-based operational semantics [4] so we shall just mention the task-specific aspects. In rule [T-FS] we use a predicate $\mathit{Running}$ that tests whether the state of a task object is currently $\mathsf{running}(\overline{F})$. We also use a function $\mathit{taskIds}(FS)$ which returns the set of all object ids o found in asynchronous frame labels $\mathsf{a}(o)$ in the frame stack FS. Thus rule [T-FS] tests that the asynchronous frame is referring to a running task, and moreover we enforce a linearity constraint, i.e. that no other asynchronous frame in the stack has the same label. We enforce this across threads in a process in rule [Proc-ok].

Theorem 1 (Preservation).

1. *If* $\Gamma; H_0 \vdash F_0\colon \phi_0 \to \phi_1$ *and* $H_0 \rhd F_0 \to H_1 \rhd F_1$ *then* $\Gamma; H_1 \vdash F_1\colon \phi_0 \to \phi_1$.
2. *If* $\vdash (H_0 \rhd FS_0)\colon \phi_0 \to \phi_1$ *and* $H_0 \rhd FS_0 \twoheadrightarrow H_1 \rhd FS_1$ *then* $\vdash (H_1 \rhd FS_1)\colon \phi_0 \to \phi_1$.

3. *If* $\vdash H_0 \triangleright P_0$ **ok** *and* $H_0 \triangleright P_0 \rightsquigarrow H_1 \triangleright P_1$ *then* $\vdash H_1 \triangleright P_1$ **ok**.

Proof. Part (1) is proved by case analysis on $H_0 \triangleright F_0 \rightarrow H_1 \triangleright F_1$. Part (2) is proved by induction on the derivation of $H_0 \triangleright FS_0 \twoheadrightarrow H_1 \triangleright FS_1$ and part (1), and part (3) by induction on the derivation of $H_0 \triangleright P_0 \rightsquigarrow H_1 \triangleright P_1$ and part (2).

Theorem 2 (Progress). *If* $\vdash (H \triangleright \langle L, s, \bar{t}\rangle^{\ell} \circ FS)\colon \texttt{void} \rightarrow \phi$ *and is in normal form, i.e. cannot transition further, then one of the following cases holds:*

1. $s = \texttt{return}\ x\texttt{;}$ *or* $s = \texttt{return;}$ *and* $FS = \epsilon$
2. $s = y\texttt{=}x\texttt{.}m\texttt{(}\bar{z}\texttt{);}$ *or* $s = y\texttt{=}x\texttt{.}f\texttt{;}$ *or* $s = x\texttt{.}m\texttt{(}\bar{z}\texttt{);}$ *and* $L(x) = \texttt{null}$
3. $s = y\texttt{=}x\texttt{.GetResult();}$, $L(x) = o$ *and* $H(o) = \langle \texttt{Task<}\tau\texttt{>}, \texttt{state} \mapsto \mathsf{running}(\overline{F})\rangle$.

Cases 1 and 2 are to be expected. Interestingly, we can rule out case 3 by simply excluding any occurrences of `GetResult` in the original program; although giving the formal details are beyond the scope of this paper. In this case, the only occurrences of `GetResult` arise from the [E-Await] transition. These frames are only resumed by the rule [E-Async-Return] which also transitions the state of the task object to $\mathsf{done}(v)$. We can also show a property that no transition rule changes the state of a task that is completed back to running. These two properties allow us to show that case 3 does not arise for `GetResult`-free source programs.

5 Extensions

5.1 Extension 1: Optimized, One-shot Semantics

The semantics presented so far is idealized: when an asynchronous frame is suspended to await a task, rule [E-Await] appends a *copy* of the frame to the task's list of waiters. At first glance, the act of copying the frame appears to require an expensive allocation of a *fresh* frame to store its contents. Notice, however, that frames are never duplicated: after copying the frame, [E-Await] pops the active frame, discarding it to proceed with its continuation, the calling stack FS. Since frames are used in a *linear* fashion, the expensive allocation on each suspend is entirely avoidable. The trick to avoiding repeated allocation is to allocate just one container for each asynchronous frame and destructively update its contents at each suspension of that frame.

The C$^\sharp$ 5.0 implementation does just this, representing a suspended frame on the heap as a "stateful" *delegate* of type `Action`. Delegates [11, Chapter 15] are just closures, containing the address of some environment and the address of some static code taking the environment as a first argument. Both addresses are immutable. The state of the frame is therefore maintained, not directly in the closure, but in its environment. To achieve this, the environment itself has mutable fields that store the current values of the frame's locals, its associated task, and the current state of the finite state machine. All read and writes of locals in the original code are compiled to indirected operations on fields of the environment. The delegate's code pointer just contains the fixed code interpreting the frame's state machine.

In this section, we formalize a high-level abstraction of this implementation. Our formalization makes the more efficient, destructive update explicit without descending all the way to the low-level representation of closures used in the concrete implementation. To do so, we require a new reference type, the delegate type `Action`. In our semantics, if not in the actual implementation, the heap representation of an action is just an object whose mutable state is a frame, containing some locals and statements. The locals map contains the *current* values of local variables. The statements represent the frame's original body in *some state of unfolding*, i.e. the frame's current "program counter". This allows us to adequately represent a paused frame, without exposing the compilation details of its encoding as a C$^\sharp$ 4.0 delegate with a fixed pointer to a mutable environment and static code. Making this change also paves the way for our formalization of the *awaitable pattern* in §5.2.

First we need to extend $\mathsf{FC}_5^\sharp$ with the `Action` delegate type and syntax for invoking an action.

$\mathsf{FC}_5^\sharp$ additional types and statements:

$\sigma ::= \ldots$	Type
`Action`	Delegate type
$s ::= \ldots$	Statement
$a\texttt{();}$	Action invocation

We also need to extend and adjust our run-time representations. `Action` is a new reference type so action values are just addresses of objects in the heap. An `Action` object, $\langle\texttt{Action}, \texttt{body} \mapsto F\rangle$, maps a mutable field `body` to a frame F, storing locals, statements and label of a suspended frame. We also need to modify tasks to track, not waiting frames ($\mathsf{running}(\overline{F})$), but waiting *actions*, represented as a list of *addresses* ($\mathsf{running}(\overline{o})$). Thus a running task will have representation $\langle\texttt{Task<}\sigma\texttt{>}, \texttt{state} \mapsto \mathsf{running}(\overline{o})\rangle$; completed tasks remains the same. The form of an asynchronous label, placed on frames, is now $\mathsf{a}(o_1, o_2)$. The new label carries not one but *two* addresses: the address of the frame's task, o_1, as before, and a second address, o_2, of an action. The action stores the previous state of the frame; recording its address in the frame label indicates where to save the next state of the frame prior to suspending.

Completing a task will need to resume a list of actions, not frames, so we adapt the definition of $resume(\overline{o})$ to set up appropriate synchronous stubs, one per action in $\overline{o}$: $resume(\overline{o}) \stackrel{\text{def}}{=} \{\langle\{x \mapsto o_i\}, x\texttt{();}\textbf{return}\texttt{;}\rangle^{\mathsf{s}} \circ \epsilon \mid o_i \in \overline{o}\}$.

Asynchronous method transition rules (One-shot semantics):

$H \rhd \langle L_0, x\texttt{();}\overline{s}\rangle^{\ell_0} \circ FS \twoheadrightarrow H \rhd \langle L_1, \overline{t}\rangle^{\ell_1} \circ \langle L_0, \overline{s}\rangle^{\ell_0} \circ FS$ [E-Action-Invoke]
where $H(L_0(x)) = \langle\texttt{Action}, \texttt{body} \mapsto \langle L_1, \overline{t}\rangle^{\ell_1}\rangle$

$H_0 \rhd \langle L_0, y_0\texttt{=}y_1\texttt{.}m\texttt{(}\overline{z}\texttt{);}\overline{s}\rangle^{\ell} \circ FS$ [E-Async-MethodOS]
$\to H_1 \rhd \langle L_1, \overline{t}\rangle^{\mathsf{a}(o_1,o_2)} \circ \langle L_0[y_0 \mapsto o_1], \overline{s}\rangle^{\ell} \circ FS$
where $H_0(L_0(y_1)) = \langle\sigma_0, FM\rangle$ and $mbody(\sigma_0, m) = mb\colon(\overline{\sigma}\ \overline{x}) \to^{\mathsf{a}} \psi$
$mb = \overline{\tau}\ \overline{y}\texttt{;}\ \overline{t}$ and $o_1, o_2 \notin dom(H), o_1 \neq o_2$
$H_1 = H_0[o_1 \mapsto \langle\psi, \texttt{state} \mapsto \mathsf{running}([\,])\rangle, o_2 \mapsto \langle\texttt{Action}, \texttt{body} \mapsto \langle L_1, \overline{t}\rangle^{\mathsf{a}(o_1,o_2)}\rangle]$
$L_1 = [\overline{x} \mapsto L_0(\overline{z}), \overline{y} \mapsto default(\overline{\tau}), \textbf{this} \mapsto L_0(y_1)]$

$H_0 \rhd \{\langle L, \texttt{return}\ y\,;\overline{s}\rangle^{\mathsf{a}(o_1,o_2)} \circ FS\} \cup P$ [E-Async-ReturnOS]
$\rightsquigarrow H_1 \rhd \{FS\} \cup \mathit{resume}(\overline{o}) \cup P$
where $H_0(o_1) = \langle \texttt{Task<}\sigma_y\texttt{>}, \texttt{state} \mapsto \mathsf{running}(\overline{o})\rangle$
$H_1 = H_0[\, o_1 \mapsto \langle \texttt{Task<}\sigma_y\texttt{>}, \texttt{state} \mapsto \mathsf{done}(L(y))\rangle,$
$o_2 \mapsto \langle \texttt{Action}, \texttt{body} \mapsto \langle L, \overline{s}\rangle^{\mathsf{a}(o_1,o_2)}\rangle]$

$H \rhd \langle L, x\texttt{=}\texttt{await}\ y\,;\overline{s}\rangle^{\mathsf{a}(o_1,o_2)} \circ FS$ [E-Await-ContinueOS]
$\twoheadrightarrow H \rhd \langle L[x \mapsto v], \overline{s}\rangle^{\mathsf{a}(o_1,o_2)} \circ FS$
where $H(L(y)) = \langle \texttt{Task<}\sigma\texttt{>}, \texttt{state} \mapsto \mathsf{done}(v)\rangle$

$H_0 \rhd \langle L, x\texttt{=}\texttt{await}\ y\,;\overline{s}\rangle^{\mathsf{a}(o_1,o_2)} \circ FS \twoheadrightarrow H_1 \rhd FS$ [E-AwaitOS]
where $L(y) = o_3, H_0(o_3) = \langle \texttt{Task<}\tau\texttt{>}, \texttt{state} \mapsto \mathsf{running}(\overline{o})\rangle$
$H_1 = H_0[\, o_3 \mapsto \langle \texttt{Task<}\tau\texttt{>}, \texttt{state} \mapsto \mathsf{running}(o_2, \overline{o})\rangle,$
$o_2 \mapsto \langle \texttt{Action}, \texttt{body} \mapsto \langle L, x\texttt{=}\ \texttt{GetResult}(y);\ \overline{s}\rangle^{\mathsf{a}(o_1,o_2)}\rangle]$

Rule [E-Action-Invoke] formalizes the invocation of an action, similar to a method call. Notice that the label of the frame is restored from the heap. In particular, an asynchronous frame will continue to signal completion through its task and have access to its action (for future suspension, if needed). Rule [E-Async-MethodOS] is similar to [E-Async-Method] but it additionally allocates a new `Action` storing the initial state of the asynchronous method. The address of the action, o_2, is recorded in the extended label of the pushed frame.

Rule [E-Async-ReturnOS] is similar to [E-Async-Return], completing the asynchronous frame's task. Though it is not necessary, we save the current locals and continuation of the `return`, $\overline{s}$, in the frame's `Action`. For this simple semantics, it should be possible to show that this action can never be invoked again.[10]

Rule [E-Await-ContinueOS] is almost identical to [E-Await-Continue], continuing execution of the current frame with the argument's result. The only difference is the extended label. There is no need to update the value of o_2 at this point. Rule [E-AwaitOS] is similar to [E-Await], but the suspend mechanism is different. This rule writes the frame's current state, locals and continuation, to its associated action, stored at address o_2, available from the frame's label. It then adds the address of that action to the incomplete task's list of waiters. Notice how the state of the action in the heap is destructively modified - there is no way to "go back" to a previous state of this frame.

Consider rule [E-AwaitOS]. It directly pushes a new asynchronous frame and assigns its task, o_1, to the caller's variable, y_0. An alternative formulation would be to push a synchronous stub that invokes the new action, o_2, and then returns the task, o_1, to the waiting caller. This would be less direct, but equivalent, and somewhat more faithful to the actual implementation. For example, the implementation of `CopyToManual` from §2.2 is essentially a stub method that, when called, invokes its internal delegate, `act()`, before returning its task.

At this point, the change to using mutable state to represent suspended frames is just an optimization. The reason is that user-code is never provided with access to a suspended frame, so the change in semantics cannot be observed.

[10] When we add support for the awaitable pattern, the potential for abuse of the awaitable protocol, will mean that this property no longer generally holds.

5.2 Extension 2: The awaitable pattern

As detailed in §2.1, in C$^\sharp$ 5.0 it is possible to await not just tasks, but values of any *awaitable* type. So far, our formalization has assumed that the only awaitable type is `Task`<σ>. In this section, we embrace the full awaitable pattern by replacing rule [C-Await] with [C-Awaitable]:

Typing rules for awaitable expressions

$$\frac{\begin{array}{c} \mathit{mtype}(\sigma_0, \mathtt{GetAwaiter}) = () \to \sigma_1 \\ \mathit{mtype}(\sigma_1, \mathtt{IsCompleted}) = () \to \mathtt{bool} \\ \mathit{mtype}(\sigma_1, \mathtt{OnCompleted}) = (\mathtt{Action}) \to \mathtt{void} \\ \mathit{mtype}(\sigma_1, \mathtt{GetResult}) = () \to \sigma_2 \end{array}}{\Gamma, x{:}\,\sigma_0 \vdash \mathtt{await}\ x{:}\,\sigma_2} \quad \text{[C-Awaitable]}$$

To avoid formalizing *properties*, we depart from C$^\sharp$ 5.0 and assume the property `IsCompleted` is an ordinary *method*; the distinction between methods and properties is entirely cosmetic so nothing is lost.

In the transition semantics `await` expressions can no longer transition atomically but must, instead, be evaluated in multiple steps. These steps commence with obtaining the argument's awaiter and proceed with calls to the awaiter's members, thus interleaving (potentially) user-defined code with the semantics of `await` construct. Rule [C-Awaitable] statically ensures that these dynamic unfoldings are well-typed.

But first, we need to arrange that tasks are awaitable and implement the remaining requirements of the awaitable pattern. Our system already provides an appropriate `GetResult` for tasks; we are left with providing `GetAwaiter`,`IsCompleted` and `OnCompleted`, ascribed with the following types:

$$\begin{aligned} \mathit{mtype}(\mathtt{Task}\texttt{<}\sigma\texttt{>}, \mathtt{GetAwaiter}) &= () \to \mathtt{Task}\texttt{<}\sigma\texttt{>} \\ \mathit{mtype}(\mathtt{Task}\texttt{<}\sigma\texttt{>}, \mathtt{IsCompleted}) &= () \to \mathtt{bool} \\ \mathit{mtype}(\mathtt{Task}\texttt{<}\sigma\texttt{>}, \mathtt{OnCompleted}) &= (\mathtt{Action}) \to \mathtt{void} \\ \mathit{mtype}(\mathtt{Task}\texttt{<}\sigma\texttt{>}, \mathtt{GetResult}) &= () \to \sigma \end{aligned}$$

To avoid hard-wiring C$^\sharp$ 5.0's generic `TaskAwaiter`<σ> type, we simplify the C$^\sharp$ 5.0 design and assume that `Task`<σ> is self-sufficient and serves as its *own* awaiter type. Correspondingly, x`.GetAwaiter()`'s type is just the type of task x; its implementation, by rule [E-Task-GetAwaiter] below, just returns the receiver.

Additional transition rules for `Task`'s awaitable operations:

$H \rhd \langle L, x\texttt{=}y\texttt{.GetAwaiter();}\overline{s}\rangle^\ell$ [E-Task-GetAwaiter]
$\to H \rhd \langle L[x \mapsto L(y)], \overline{s}\rangle^\ell$ where $H(L(y)) = \langle \mathtt{Task}\texttt{<}\sigma\texttt{>}, \mathit{FM}\rangle$

$H \rhd \langle L, x\texttt{=}y\texttt{.IsCompleted();}\overline{s}\rangle^\ell$ [E-Task-IsCompleted]
$\to H \rhd \langle L[x \mapsto \mathtt{true}], \overline{s}\rangle^\ell$ where $H(L(y)) = \langle \mathtt{Task}\texttt{<}\sigma\texttt{>}, \mathtt{state} \mapsto \mathsf{done}(v)\rangle$
$\to H \rhd \langle L[x \mapsto \mathtt{false}], \overline{s}\rangle^\ell$ where $H(L(y)) = \langle \mathtt{Task}\texttt{<}\sigma\texttt{>}, \mathtt{state} \mapsto \mathsf{running}(\overline{o})\rangle$

$H_0 \rhd \langle L, x\texttt{.OnCompleted(}y\texttt{);}\overline{s}\rangle^\ell$ [E-Task-OnCompleted-Suspend]
$\to H_1 \rhd \langle L, \overline{s}\rangle^\ell$
where $L(x) = o_1$ and $H_0(o_1) = \langle \mathtt{Task}\texttt{<}\sigma\texttt{>}, \mathtt{state} \mapsto \mathsf{running}(\overline{o})\rangle$
$L(y) = o_2$ and $H_1 = H_0[o_1 \mapsto \langle \mathtt{Task}\texttt{<}\sigma\texttt{>}, \mathtt{state} \mapsto \mathsf{running}(o_2, \overline{o})\rangle$

$H \rhd \{\langle L, x.\mathtt{OnCompleted}(y);\overline{s}\rangle^{\ell} \circ FS\} \cup P$ [E-Task-OnCompleted-Resume]
$\rightsquigarrow H \rhd \{\langle L, \overline{s}\rangle^{\ell} \circ FS\} \cup resume(o) \cup P$
where $H(L(x)) = \langle \mathtt{Task{<}\sigma{>}}, \mathtt{state} \mapsto \mathsf{done}(v)\rangle$ and $L(y) = o$

Task's implementation of IsCompleted() tests the state field of the receiver, returning true if and only if it is done(_). The implementation of OnCompleted(y) adds its callback y (an Action), to the receiver's list of waiters. If the task is already completed, the action cannot be stored and must, instead, be resumed in the process. The latter rule is required since there is a race between testing that a task IsCompleted(), finding it is false, and calling OnCompleted(y) - some other thread could intervene and complete the task *before* OnCompleted(y) executes.

We can now formalize the operational semantics of await on any awaitable. Because we need to interleave the execution of methods from the awaitable pattern – which take several transitions and could be user-defined – with the semantics of await, we need to introduce two additional, transient *control* statements that can only appear within asynchronous frames.

FC$^{\sharp}_{5}$ additional control statements:

$s ::= \ldots$	Statement
suspend;	suspend
getcc(Action a){$\overline{s}$};	get current continuation

Though artificial, these statements have direct interpretations as intermediate steps of a compiler generated finite-state-machine.[11]

We define $unfold(x = \mathtt{await}\ y; \overline{s})^{(z,b)}$ to be the syntactic unfolding of an await as a new sequence of statements using temporaries z and b (note a is bound):

$$unfold(x = \mathtt{await}\ y; \overline{s})^{(z,b)} \stackrel{\text{def}}{=} \begin{cases} z = y\mathtt{.GetAwaiter();} \\ b = z\mathtt{.IsCompleted();} \\ \mathtt{if}\ (b)\ \{\} \\ \mathtt{else}\ \{ \\ \ \mathsf{getcc}(\mathtt{Action}\ a)\{z\mathtt{.OnCompleted}(a)\mathtt{;}\mathsf{suspend}\mathtt{;}\}\mathtt{;} \\ \} \\ x = z\mathtt{.GetResult();} \\ \overline{s} \end{cases}$$

The operation unfolds an await of an awaitable object y by first retrieving its awaiter z and setting b to determine if the awaiter is complete. If complete, the code falls through the conditional. If incomplete, the code transfers the current continuation of the getcc statement to z (through a) and suspends. The continuation of both the true branch and the getcc statement is just $x = z\mathtt{.GetResult();}\overline{s}$. It assigns the result of the awaiter to x, and proceeds with the original continuation $\overline{s}$ of the await.

[11] For example, in our hand-coded CopyToManual from §2.2, suspend corresponds to a return; from the act delegate that pauses execution (states 1 and 2 of the switch); getcc(Action a){$\overline{s}$}; corresponds to advancing the (shared) state variable to the next logical state (following getcc(Action a){$\overline{s}$};) and then reading the value of the task's state machine (the delegate act).

Awaitable pattern, asynchronous method transition rules:

$H_0 \triangleright \langle L_0, x\texttt{=}\texttt{await}\ y\texttt{;}\overline{s}\rangle^{\mathsf{a}(o_1,o_2)}$ **[E-Awaitable]**
$\rightarrow H_0 \vdash \langle L_1, \mathit{unfold}(x \texttt{ = } \texttt{await}\ y; \overline{s})^{(z,b)}\rangle^{\mathsf{a}(o_1,o_2)}$
where $L_1 = L_0[z \mapsto \texttt{null}, b \mapsto \texttt{false}]$ and $z, b \notin \mathit{dom}(L_0), z \neq b$

$H_0 \triangleright \langle L, \mathsf{getcc}\texttt{(Action}\ a\texttt{)\{}\overline{s}\texttt{\};}\overline{t}\rangle^{\mathsf{a}(o_1,o_2)}$ **[E-Get-Current-Continuation]**
$\rightarrow H_1 \triangleright \langle L[a \mapsto o_2], \overline{s}\rangle^{\mathsf{a}(o_1,o_2)}$
where $a \notin \mathit{dom}(L)$ and $H_1 = H_0[o_2 \mapsto \langle \texttt{Action}, \mathsf{body} \mapsto \langle L, \overline{t}\rangle^{\mathsf{a}(o_1,o_2)}\rangle]$

$H \triangleright \langle L, \mathsf{suspend}\texttt{;}\overline{s}\rangle^{\mathsf{a}(o_1,o_2)} \circ FS \twoheadrightarrow H \triangleright FS$ **[E-Suspend]**

Rule [E-Awaitable] syntactically unfolds its await expression using fresh temporaries, z and b. In rule [E-Get-Current-Continuation], getcc(`Action` a)\{$\overline{s}$\} unfolds by first saving its continuation $\overline{t}$ in the frame's task, o_2, discarding it from the active frame, and then entering the body $\overline{s}$. When $\overline{s}$ is just z`.OnCompleted(`a`);`suspend`;`, as per rule [E-Awaitable], $\overline{s}$ will *transfer* the current continuation to the awaiter and pause execution.

In rule [E-Suspend] the suspend control statement pauses the asynchronous frame. This is similar to a `return`, but the frame's task is not marked completed and remains in its current state. One might expect this state to be running(_) but it may not be, depending on the semantics of `OnCompleted`.

Although our semantics unfolds `await`s dynamically, it would also be possible to statically expand well-typed `await` expressions by a source-to-source translation, sketched here:

$$
\begin{array}{lll}
[x = \texttt{await}\ y; \overline{s_0}]^{\Gamma} & \stackrel{\text{def}}{=} & (\Gamma_1, \overline{s_2}) \text{ where} \\
 & & \mathit{mtype}(\Gamma(y), \texttt{GetAwaiter}) = () \rightarrow \sigma_1 \\
 & & (\Gamma_1, \overline{s_1}) = [\overline{s_0}]^{\Gamma, z:\sigma_1, b:\texttt{bool}} \\
 & & \overline{s_2} = \mathit{unfold}(x \texttt{ = } \texttt{await}\ y; \overline{s_1})^{(z,b)} \\
 & & b, z \notin \mathit{dom}(\Gamma), b \neq z \\
[s; \overline{s}]^{\Gamma} & \stackrel{\text{def}}{=} & \ldots
\end{array}
$$

The translation must be type-directed (in order to determine awaiter types) and needs to produce a new context as well as the list of statements in order to properly account for generated variables. Notice, however, that it is finite and does not need to duplicate the input continuation $\overline{s_0}$, making it suitable for compile-time expansion.

`OnCompleted`'s *one-shot* restriction, explained. Once we add the awaitable pattern to the mix, the optimization described in §5.1 becomes a proper change to the semantics, with observable consequences. The culprit is the awaitable pattern's `OnCompleted(`a`)` method since it provides user-code with access to the *one-shot* continuation, a, of the frame, represented not as a pure value but as a stateful object. Recall that our informal description of the awaiter pattern stipulated that implementations of `OnCompleted` are required to invoke their action *at most once*. The reason why should now be clear. Invoking the action will resume the frame and potentially modify the action's state. In our semantics, the update would happen at the next suspension. In the real implementation, the update would happen at the next write to some notionally local, but actually shared, variable of the frame. Two concurrent invocations of the same action could lead to unpredictable behaviour, as each races to save its next, possibly different state in the very same action. Frame execution depends on the shared heap, modified non-deterministically by rule [E-Schedule], so two invocations could very easily reach different states.

As it happens, when extended with the awaitable pattern, even the simpler, copying semantics cannot tolerate multiple invocations of a continuation, but the reason is more subtle. In the copying semantics, even a copy of a frame is inherently stateful because its label will contain a reference to the original frame's task. Allocated on the heap, this task is shared state: several invocations of the same continuation would race, with possibly different results, to complete the very same task on exit. Part of the semantics of tasks is that they should complete at most once. This invariant is violated by any abuse of one-shot continuations, as enabled by the awaitable pattern.

5.3 Optimizing Asynchronous Tail Calls

Our semantics gives us a model of space usage, suitable for proposing space optimizations. One effective optimization is tail call elimination. For ordinary methods, a tail call is just a synchronous method call followed by an immediate return of its result:

$$x\texttt{=}y\,\texttt{.}\,m\texttt{(}\overline{z}\texttt{)};\,\texttt{return}\;x\,\texttt{;}$$

The tail call *optimization* is to pop the current active frame before pushing the callee's frame, thus arranging for the callee to return on behalf of the current method. The space *saved* is the popped frame, plus any heap dedicated to that frame.

Asynchronous methods also support a natural notion of tail call. An asynchronous method call followed by the return of its awaited result is an *asynchronous tail call*:

$$x\texttt{=}y\,\texttt{.}\,m\texttt{(}\overline{z}\texttt{)}\,\texttt{;}\,w\;\texttt{=}\;\texttt{await}\;x;\,\texttt{return}\;w\,\texttt{;}$$

Typing requires that x an w have the *same* type and that m and the enclosing method (which must be asynchronous due to the `await`) have the same task type.

We suggest the following optimization of asynchronous tail calls:

Asynchronous Tail Call Optimization:

$$H_0 \rhd \langle L_0, y_0\texttt{=}y_1\,\texttt{.}\,m\texttt{(}\overline{z}\texttt{)}\,\texttt{;}\,y_2\;\texttt{=}\;\texttt{await}\;y_0;\,\texttt{return}\;y_2\,\texttt{;}\,\overline{s}\rangle^{\mathsf{a}(o_1,o_2)} \circ FS \qquad \text{[E-Async-TailCallOS]}$$
$$\rightarrow H_1 \rhd \langle L_1, \overline{t}\rangle^{\mathsf{a}(o_1,o_3)} \circ FS$$
$$\begin{aligned}
\text{where } & H_0(L_0(y_1)) = \langle \sigma_0, FM\rangle \\
& mbody(\sigma_0, m) = mb\colon (\overline{\sigma}\;\overline{x}) \rightarrow^{\mathsf{a}} \psi \\
& mb = \overline{\tau}\;\overline{y}\,\texttt{;}\;\overline{t} \\
& o_3 \notin dom(H) \cup \{o_1, o_2\} \\
& H_1 = H_0[o_2 \mapsto \langle \texttt{Action}, \texttt{body} \mapsto \langle L_0, \overline{s}\rangle^{\mathsf{a}(o_1,o_2)}\rangle, \\
& \qquad\qquad o_3 \mapsto \langle \texttt{Action}, \texttt{body} \mapsto \langle L_1, \overline{t}\rangle^{\mathsf{a}(o_1,o_3)}\rangle] \\
& L_1 = [\overline{x} \mapsto L_0(\overline{z}), \overline{y} \mapsto default(\overline{\tau}), \texttt{this} \mapsto L_0(y_1)]
\end{aligned}$$

This optimization works by re-using the task of the current frame (the destination for its result) as the task for the newly pushed frame, avoiding the allocation of a fresh one. As in ordinary tail-call elimination, the current frame can be eagerly popped, saving space. Since we do not model garbage collection, this rule diligently updates the action, o_2, of the current frame, despite it having (notionally) exited. However, a collector should be able to free the storage associated with o_2. To see why, observe that o_2 should not be in any other task's list of waiters (since its frame was executing). This means that the only reference to o_2 is the recursive one from its own stored frame. Hence o_2 is unreachable.

The C♯ 5.0 implementation does not currently apply this tail call optimization; doing so would require a minor change in compilation strategy: passing an additional, possibly `null`, destination task to asynchronous method calls. This would allow a caller to optionally transfer its own task, when appropriate at tail calls, and a callee to allocate a fresh task, only if its incoming task is `null`.

5.4 Extension 3: Exceptions

Reasoning about control flow in the presence of exceptions in C♯ is challenging; the addition of asynchrony further complicates the situation. In this section we extend $\mathsf{FC}_5^\sharp$ with exceptions and give a semantics that clearly lays out how exceptions affect control flow within functions, between function calls, and between different threads of control.

$\mathsf{FC}_5^\sharp$ additional statements for exceptions:

$s ::= \ldots$	Statement	
$\texttt{throw}\ x\texttt{;}$		Throw statement
$\texttt{try}\ \{\overline{s}\}\ \texttt{catch}\ (\texttt{C}\ x)\{\overline{t}\}\ \texttt{finally}\ \{\overline{u}\}\texttt{;}$		Try block

A `throw` statement can be type checked at any type. The type of the body of a `try` statement must match the types of the body of the associated `catch` clause, but the body of the associated `finally` clause may have any type. Note that well-formed `try` blocks may not attempt to return from within the `finally` clause, and a catch clause may not specify a class to catch that is an instance of a class specified in an earlier clause, i.e., there are no "dead" catch clauses.

$\mathsf{FC}_5^\sharp$ additional statement type checking for exceptions:

$$\frac{}{\Gamma, x\colon \sigma \vdash \texttt{throw}\ x\texttt{;}\colon \tau}\quad \text{[C-Throw]}$$

$$\frac{\Gamma \vdash \overline{s}\colon \sigma \qquad \Gamma, x\colon \texttt{C} \vdash \overline{t}\colon \sigma \qquad \Gamma \vdash \overline{u}\colon \tau}{\Gamma \vdash \texttt{try}\ \{\overline{s}\}\ \texttt{catch}\ (\texttt{C}\ x)\{\overline{t}\}\ \texttt{finally}\ \{\overline{u}\}\colon \sigma}\quad \text{[C-Try]}$$

We track exceptional control flow in the operational semantics by adding an *exception frame stack* to every frame stack. A frame is now of the form $\langle L, \overline{s}, XS\rangle^\ell$ where XS is an exception frame stack of the form $X_1 \circ \cdots X_n \circ \epsilon$, where the X_i are exception frames which are of the forms $\mathsf{catch}(\texttt{C}, y, \overline{t})$, $\mathsf{pop}(x)$, $\mathsf{finally}(\overline{u})$, $\mathsf{throw}(o)$, $\mathsf{return}(\overline{v})$, and $\mathsf{kont}(\overline{s})$.

Except for the [E-Return] and [E-Return-Val] rules, the simple transition rules given earlier remain essentially the same; they do not modify the exception frame stack. The new rules for [E-Return] and [E-Return-Val] as well as the rules for handling `throw` and `try` statements are as follows.

Simple transition rules:

$H \triangleright \langle L, \texttt{try}\ \{\overline{s}\}\ \texttt{catch}\ (\texttt{C}\ y)\{\overline{t}\}\ \texttt{finally}\ \{\overline{u}\};\overline{s_k}, XS\rangle^\ell$ [E-Try]
$\rightarrow H \triangleright \langle L, \overline{s}, \mathsf{catch}(\texttt{C}, y, \overline{t}) \circ \mathsf{finally}(\overline{u}) \circ \mathsf{kont}(\overline{s_k}) \circ XS\rangle^\ell$

$H \triangleright \langle L, \texttt{throw}\ x;\overline{s}, XS\rangle^\ell \rightarrow H \triangleright \langle L, \epsilon, \mathsf{throw}(L(x)) \circ XS\rangle^\ell$ [E-Throw]

$H \triangleright \langle L, \texttt{return};\overline{s}, XS\rangle^\ell \rightarrow H \triangleright \langle L, \epsilon, \mathsf{return}() \circ XS\rangle^\ell$ [E-Return]

$H \triangleright \langle L, \texttt{return}\ x;\overline{s}, XS\rangle^\ell \rightarrow H \triangleright \langle L, \epsilon, \mathsf{return}(L(x)) \circ XS\rangle^\ell$ [E-Return-Val]

The handling of all control flow, both usual and exceptional, is performed by transition rules that operate on the exception frame stack. These rules are triggered only when there are no remaining statements to be executed. Note that because a well-formed `finally` clause may not contain a `return` statement, we are guaranteed to never encounter an exception frame stack of the form $\mathsf{return}(\overline{v}) \circ \mathsf{return}(\overline{v}') \circ XS$ or of the form $\mathsf{return}(\overline{v}) \circ \mathsf{throw}(o) \circ XS$ so there are no transitions that handle these cases.

Each transition decreases the number of frames on the exception stack. Since well-formed functions must execute a `return` statement, the frame transition relation $\rightarrow$ steps until the exception frame stack contains a single $\mathsf{return}(\overline{v})$ or single $\mathsf{throw}(o)$ exception frame and there are no more statements to execute. At this point the synchronous or asynchronous method transition rules kick in, depending on whether the topmost frame is synchronous or asynchronous.

Simple transition rules for the exception frame stack:

$H \triangleright \langle L, \epsilon, \mathsf{catch}(\texttt{C}, y, \overline{t}) \circ XS\rangle^\ell \rightarrow H \triangleright \langle L, \epsilon, XS\rangle^\ell$ [E-Catch]

$H \triangleright \langle L, \epsilon, \mathsf{pop}(y) \circ XS\rangle^\ell \rightarrow H \triangleright \langle L \setminus \{y\}, \epsilon, XS\rangle^\ell$ [E-Pop]

$H \triangleright \langle L, \epsilon, \mathsf{finally}(\overline{u}) \circ XS\rangle^\ell \rightarrow H \triangleright \langle L, \overline{u}, XS\rangle^\ell$ [E-Finally]

$H \triangleright \langle L, \epsilon, \mathsf{throw}(o) \circ \mathsf{catch}(\texttt{C}, y, \overline{t}) \circ XS\rangle^\ell$ [E-ThrowCatch]
$\rightarrow H \triangleright \langle L[y \mapsto o], \overline{t}, \mathsf{pop}(y) \circ XS\rangle^\ell$ where $H(o) = \langle \sigma, FM\rangle$ and $\sigma = \texttt{C}$

$H \triangleright \langle L, \epsilon, \mathsf{throw}(o) \circ \mathsf{catch}(\texttt{C}, y, \overline{t}) \circ XS\rangle^\ell$ [E-ThrowNoCatch]
$\rightarrow H \triangleright \langle L, \epsilon, \mathsf{throw}(o) \circ XS\rangle^\ell$ where $H(o) = \langle \sigma, FM\rangle$ and $\sigma \neq \texttt{C}$

$H \triangleright \langle L, \epsilon, \mathsf{throw}(o) \circ \mathsf{pop}(y) \circ XS\rangle^\ell \rightarrow H \triangleright \langle L \setminus \{y\}, \epsilon, \mathsf{throw}(o) \circ XS\rangle^\ell$ [E-ThrowPop]

$H \triangleright \langle L, \epsilon, \mathsf{throw}(o) \circ \mathsf{finally}(\overline{u}) \circ XS\rangle^\ell \rightarrow H \triangleright \langle L, \overline{u}, \mathsf{throw}(o) \circ XS\rangle^\ell$ [E-ThrowFinally]

$H \triangleright \langle L, \epsilon, \mathsf{throw}(o_1) \circ \mathsf{throw}(o_2) \circ XS\rangle^\ell \rightarrow H \triangleright \langle L, \epsilon, \mathsf{throw}(o_1) \circ XS\rangle^\ell$ [E-ThrowThrow]

$H \triangleright \langle L, \epsilon, \mathsf{throw}(o) \circ \mathsf{return}(\overline{v}) \circ XS\rangle^\ell \rightarrow H \triangleright \langle L, \epsilon, \mathsf{throw}(o) \circ XS\rangle^\ell$ [E-ThrowReturn]

$H \triangleright \langle L, \epsilon, \mathsf{throw}(o) \circ \mathsf{kont}(\overline{s}) \circ XS\rangle^\ell \rightarrow H \triangleright \langle L, \epsilon, \mathsf{throw}(o) \circ XS\rangle^\ell$ [E-ThrowCont]

$H \triangleright \langle L, \epsilon, \mathsf{return}(\overline{v}) \circ \mathsf{catch}(\texttt{C}, y, \overline{t}) \circ XS\rangle^\ell \rightarrow H \triangleright \langle L, \epsilon, \mathsf{return}(\overline{v}) \circ XS\rangle^\ell$ [E-ReturnCatch]

$H \triangleright \langle L, \epsilon, \mathsf{return}(\overline{v}) \circ \mathsf{pop}(y) \circ XS\rangle^\ell \rightarrow H \triangleright \langle L \setminus \{y\}, \epsilon, \mathsf{return}(\overline{v}) \circ XS\rangle^\ell$ [E-ReturnPop]

$H \rhd \langle L, \epsilon, \textsf{return}(\overline{v}) \circ \textsf{finally}(\overline{u}) \circ XS \rangle^{\ell} \rightarrow H \rhd \langle L, \overline{u}, \textsf{return}(\overline{v}) \circ XS \rangle^{\ell}$ [E-ReturnFinally]

$H \rhd \langle L, \epsilon, \textsf{return}(\overline{v}) \circ \textsf{kont}(\overline{s}) \circ XS \rangle^{\ell} \rightarrow H \rhd \langle L, \epsilon, \textsf{return}(\overline{v}) \circ XS \rangle^{\ell}$ [E-ReturnCont]

$H \rhd \langle L, \epsilon, \textsf{kont}(\overline{s}) \circ XS \rangle^{\ell} \rightarrow H \rhd \langle L, \overline{s}, XS \rangle^{\ell}$ [E-Cont]

The synchronous method transition rules handle returning from and throwing from within a synchronous frame. Note that an exception propagates up the frame stack by immediately ceasing the execution of statements in the enclosing frame and moving to transitioning on the exception frame stack.

Synchronous method transition rules:

$H \rhd \langle L_0, \epsilon, \textsf{return}(v) \circ \epsilon \rangle^{\mathsf{s}} \circ \langle L_1, \overline{t}, XS \rangle^{\ell}_{x} \circ FS$ [E-Sync-Return-Val]
$\twoheadrightarrow H \rhd \langle L_1[x \mapsto v], \overline{t}, XS \rangle^{\ell} \circ FS$

$H \rhd \langle L_0, \epsilon, \textsf{return}() \circ \epsilon \rangle^{\mathsf{s}} \circ \langle L_1, \overline{t}, XS \rangle^{\ell} \circ FS \twoheadrightarrow H \rhd \langle L_1, \overline{t}, XS \rangle^{\ell} \circ FS$ [E-Sync-Return]

$H \rhd \langle L_0, \epsilon, \textsf{throw}(o) \circ \epsilon \rangle^{\mathsf{s}} \circ \langle L_1, \overline{t}, XS \rangle^{\ell} \circ FS$ [E-Sync-Throw]
$\twoheadrightarrow H \rhd \langle L_1, \epsilon, \textsf{throw}(o) \circ XS \rangle^{\ell} \circ FS$

Returning from an asynchronous frame sets the frame's associated `Task`'s state to $\textsf{done}()$, as before. We add a new task state, $\textsf{failed}(o)$, that corresponds to a thrown exception. The $\textsf{failed}(o)$ task state stores a heap reference to the thrown object so that the exception can be propagated to the awaiting frame, and the [E-Task-GetResult-Failure] rule performs the propagation.

Asynchronous method transition rules:

$H_0 \rhd \{\langle L, \epsilon, \textsf{return}(v) \circ \epsilon \rangle^{\mathsf{a}(o)} \circ FS\} \cup P$ [E-Async-Return]
$\rightsquigarrow H_1 \rhd \{FS\} \cup \mathit{resume}(\overline{F}) \cup P$
where $H_0(o) = \langle \texttt{Task<}\sigma\texttt{>}, \texttt{state} \mapsto \textsf{running}(\overline{F}) \rangle$
$H_1 = H_0[o \mapsto \langle \texttt{Task<}\sigma\texttt{>}, \texttt{state} \mapsto \textsf{done}(v) \rangle]$

$H_0 \rhd \{\langle L, \epsilon, \textsf{throw}(o_1) \circ \epsilon \rangle^{\mathsf{a}(o_2)} \circ FS\} \cup P$ [E-Async-Throw]
$\rightsquigarrow H_1 \rhd \{FS\} \cup \mathit{resume}(\overline{F}) \cup P$
where $H_0(o_2) = \langle \texttt{Task<}\sigma\texttt{>}, \texttt{state} \mapsto \textsf{running}(\overline{F}) \rangle$
$H_1 = H_0[o_2 \mapsto \langle \texttt{Task<}\sigma\texttt{>}, \texttt{state} \mapsto \textsf{failed}(o_1) \rangle]$

$H \rhd \langle L, x\texttt{=}y\texttt{.Result();}\overline{s}, XS \rangle^{\ell}$ [E-Task-Result-Failure]
$\rightarrow H \rhd \langle L, \epsilon, \textsf{throw}(o) \circ XS \rangle^{\ell}$
where $H(L(y)) = \langle \texttt{Task<}\sigma\texttt{>}, \texttt{state} \mapsto \textsf{failed}(o) \rangle$

$H \rhd \langle L, x\texttt{=GetResult}(y)\texttt{;}\overline{s}, XS \rangle^{\ell}$ [E-Task-GetResult-Failure]
$\rightarrow H \rhd \langle L, \epsilon, \textsf{throw}(o) \circ XS \rangle^{\ell}$
where $H(L(y)) = \langle \texttt{Task<}\sigma\texttt{>}, \texttt{state} \mapsto \textsf{failed}(o) \rangle$

6 Related work

The debate regarding how asynchronous software should be structured is both old and ongoing. Lauer and Needham [13] noted that the thread-based and event-based models are dual; a program written in one style can be transformed into a program written in the other style. Though this establishes that the two models are equivalent in expressive power, it does not resolve the question of which model is easier to use or reason about.

Ousterhout [15] famously stated that "threads are a bad idea (for most programs)." His argument revolves around the claim that threads are more difficult to program than events because the programmer must reason about shared state, locks, race conditions, etc., and that they are only necessary when true concurrency–in contrast to asynchrony—is desired. Though he conflates the threaded model of programming, in which there is no inversion of control, with concurrency, his observation that the programmer should be able to reason about the operation of code is well-taken.

SEDA [23] demonstrates that the event model can be highly scalable. Servers designed using SEDA are broken into separate stages with associated event queues. Each stage automatically tunes its resource usage and computation to meet application-wide performance goals. Within each stage multiple threads may process events, but these threads are utilized only for concurrency. The programmer still has to manually manage the state associated with each event. SEDA's goal is to provide an architecture that automatically tunes resource usage and adapts to overload conditions, not to make programming servers easier.

The dual argument in favor of threads over events is made by von Behren et al. [20]. They tease apart the different aspects of threads that may make them undesirable and argue that most of these deficiencies are merely implementation artifacts. Capriccio [21] demonstrates that this is the case by providing a very efficient cooperative threading mechanism that avoids inversion of control and provides an efficient runtime. Because it uses cooperative threading, Capriccio avoids the overhead of concurrency, and code transformations to insert stack checks allow threads' stacks to grow without requiring large amounts of pre-allocated stack space. Like Capriccio, asynchronous $C^\sharp$ allows programs to be written in a natural way while providing an efficient implementation. However, instead of attempting to provide a general cooperative threading mechanism, it permits programmers to write asynchronous code in a straight-line fashion by automating stack-ripping [1] via compilation to a state machine. Like the state machine translation $C^\sharp$ uses to implement asynchrony, it is reminiscent of coroutine implementations [19, 7] built on Duff's device [6] that do not attempt to maintain local state across state machine transitions.

The observation that continuations provide a natural substrate on which to build a threading mechanism was made by Wand [22]. The observation that more restrictive, but more efficient, one-shot continuations suffice for continuation based threading dates back to [5]. Li and Zdancewic [14] use continuations to unify the event and thread-based models. They leverage the continuation monad in Haskell to allow programmers to write straight line code that is desugared into continuation passing style (CPS), thus allowing it to be used in an event-based IO framework they construct.

The *computation expressions* [18] of $F^\sharp$ provide a generalized monadic syntax, which is essentially an extended form of Haskell's do-notation. When specialised to $F^\sharp$'s *asynchronous workflow* monad—itself a continuation monad—they allow programmers to write monadic code that is syntactically expanded to explicit continuation-passing-code. This offers much of the legibility of programming in direct-style while, at the same time, providing access to the implicit continuations (as F$^\sharp$ functions) whenever required (e.g. when supplying callbacks to asynchronous calls). The generality of computation expressions has a cost: each continuation of a monadic let is a heap-allocated function; and every wait on an asynchronous value typically requires an expensive allocation of a fresh continuation. This is similar to our idealized semantics in §3. The upshot is that these continuations can, in principle, be invoked several times, allowing the encoding of a much wider range of control operators than the one-shot actions of C$^\sharp$'s feature. But there are more differences. In F$^\sharp$, computation expressions produce *inert* values that are easily composed but must be explicitly run to produce a result. In C$^\sharp$, on the other hand, each task returned by an async method call represents a *running* computation. This makes it easier to initiate asynchronous work but, perhaps, harder to define combinators that compose asynchronous methods. Thus, though inspired by F$^\sharp$'s asynchronous workflows, C$^\sharp$ 5.0 support for asynchrony is quite different in performance, expressivity and usage.

Scala actors [10] provide an asynchronous, message passing model for programming with concurrent processes. Asynchronous programs may be written in terms of *receive*, which suspends the current thread until a message is received, or *react*, which enqueues a continuation that is called when a message is received; *receive* provides a thread-based interface and *react* provides an event-based interface to an underlying message passing framework. Although the two programming models are made similar through the use of various combinators, the programmer must still significantly modify code to move from one style to the other. Rompf et al. [17] use a type and effect system to selectively CPS convert Scala programs, providing a less onerous path from threads to event-based asynchronous code, but C$^\sharp$ 5.0's `async` keyword is even more lightweight.

Despite Ousterhout's early admonition that reasoning about threads is difficult and error-prone, none of the work mentioned makes an explicit attempt to provide programmers with a set of reasoning principles for asynchronous code. Although we believe C$^\sharp$'s support for asynchrony exists at a useful point in the design space, our focus is on providing these reasoning principles. Threads or events, manual stack ripping or CPS, a programmer must have clear ways to reason about code behavior in order to build correct systems of any kind.

7 Conclusions

Real-world software construction demands effective methods for dealing with asynchrony. For such a method to be termed "effective," it must not require large-scale, manual code transformations such as stack ripping. Sequential computations should be expressible with sequential code, even if individual operations may execute asynchronously. Splitting sequential code up into a series of callbacks or explicitly rewriting it as a state machine is a steep price to pay, making code difficult to write, difficult to read, and difficult to reason about; if in doubt, contrast the synchronous, `async`-enabled, and hand-written state machine versions of the stream copying function from §2.2. While previous work has provided syntactic and library support for dealing with asynchrony, C$^\sharp$ 5.0 brings this support to a widely-deployed, mainstream language.

One deficiency of this previous work is a lack of reasoning principles for asynchronous code. Our primary contribution is an operational semantics for C$^\sharp$ 5.0 that allows programmers to answer questions about the code they write and make conclusions about the impact of adding asynchrony to their code. For example, using our semantics, the programmer can see that calling an `async` method does not spawn a new thread, but instead executes the method on the current stack. With the optimized semantics in §5.1, one can even begin to reason about space usage by, e.g., observing that the state of an `async` method is always stored in the same heap-allocated `Action`, which requires only a single allocation.

We plan to continue our formalization of C$^\sharp$ 5.0 by incorporating additional language features, such as cancellation tokens and synchronization context object. Our semantics have been translated to Coq. We will use this as a foundation to validate a translation from Featherweight C$^\sharp$ 5.0—including the `async` construct—to Featherweight CIL, an idealized version of the bytecode targeted by the C$^\sharp$ 5.0 compiler. Validation of this translation will prove that programmers can reason in terms of our high-level operational sematics even though the high-level program has been translated to bytecode and it is the bytecode that is actually executed.

While language support is important for easing the pain of writing asynchronous code, a corresponding semantics is vital for writing correct software. With our semantics, C$^\sharp$ 5.0 both eases the pain and provides the necessary tools for thinking carefully about the remedy.

Acknowledgements We thank the C$^\sharp$ and Visual Basic design teams for their collaboration, especially Lucian Wischik who led much of the design and implementation effort.

References

1. A. Adya, J. Howell, M. Theimer, W. J. Bolosky, and J. R. Douceur. Cooperative task management without manual stack management. In *Proceedings of USENIX*, 2002.
2. G. Bierman, E. Meijer, and M. Torgersen. Lost in translation: Formalizing proposed extensions to C$^\sharp$. In *Proceedings of OOPSLA*, 2007.
3. G. Bierman, E. Meijer, and M. Torgersen. Adding dynamic types to C$^\sharp$. In *Proceedings of ECOOP*, 2010.
4. G. Bierman, M. Parkinson, and A. Pitts. MJ: An imperative core calculus for Java and Java with effects. Technical Report 563, University of Cambridge Computer Laboratory, 2003.
5. C. Bruggeman, O. Waddell, and R. K. Dybvig. Representing control in the presence of one-shot continuations. In *Proceedings of PLDI*, 1996.
6. T. Duff. Re: Explanation, please!, Aug. 1988. USENET Article.
7. A. Dunkels, O. Schmidt, T. Voigt, and M. Ali. Protothreads: Simplifying Event-Driven programming of Memory-Constrained embedded systems. In *Proceedings of SenSys*, 2006.
8. C. Flanagan, A. Sabry, B. Duba, and M. Felleisen. The essence of compiling with continuations. In *Proceedings of PLDI*, 1993.
9. M. Flatt, S. Krishnamurthi, and M. Felleisen. A programmer's reduction semantics for classes and mixins. Technical Report TR-97-293, Rice University, 1997.
10. P. Haller and M. Odersky. Scala actors: Unifying thread-based and event-based programming. *Theoretical Computer Science*, 410(2-3):202–220, Feb. 2009.

11. A. Hejlsberg, M. Torgersen, S. Wiltamuth, and P. Golde. *The C$^\sharp$ Programming Language*. Addison-Wesley, fourth edition, 2011.
12. A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM TOPLAS*, 23(3):396–450, 2001.
13. H. C. Lauer and R. M. Needham. On the duality of operating system structures. *Operating Systems Review*, 13(2):3–19, 1979.
14. P. Li and S. Zdancewic. Combining events and threads for scalable network services. In *Proceedings of PLDI*, 2007.
15. J. K. Ousterhout. Why threads are a bad idea (for most purposes). In *USENIX Winter Technical Conference*, June 1996. Invited Talk.
16. B. Pierce and D. Turner. Local type inference. In *Proceedings of POPL*, 1998.
17. T. Rompf, I. Maier, and M. Odersky. Implementing first-class polymorphic delimited continuations by a type-directed selective CPS-transform. In *Proceedings of ICFP*, 2009.
18. D. Syme, T. Petricek, and D. Lomov. The F$^\sharp$ asynchronous programming model. In *Proceedings of PADL*, 2011.
19. S. Tatham. Coroutines in C. `http://www.chiark.greenend.org.uk/~sgtatham/coroutines.html`, 2000.
20. R. von Behren, J. Condit, and E. Brewer. Why events are a bad idea (for high-concurrency servers). In *Proceedings of HotOS*, 2003.
21. R. von Behren, J. Condit, F. Zhou, G. C. Necula, and E. Brewer. Capriccio: scalable threads for internet services. In *Proceedings of SOSP*, 2003.
22. M. Wand. Continuation-based multiprocessing. In *Proceedings of LISP and functional programming*, 1980.
23. M. Welsh, D. Culler, and E. Brewer. SEDA: an architecture for well-conditioned, scalable internet services. In *Proceedings of SOSP*, 2001.