Pause ’n’ play: Formalizing asynchronous C♯

Gavin Bierman

Microsoft Research

Joint work with C. Russo, G. Mainland, E. Meijer and M. Torgersen

ECOOP, June 2012

What problems do developers face today?

- Want “fluid” apps
- i.e. no hourglass/animated circle cursor!

Heart of the problem: **Synchronous operations**

- Simple to use but:
  - prevent progress until done,
  - reveal latency,
  - waste resources (calling threads).

What did my professor teach me?

Use **asynchronous** operations instead:

- enable concurrent progress whilst operations are running,
- hide latency,
- free up resources (calling threads).

Asynchronous coding has always been possible in C♯...
...but it's never been easy.

Today's message: C♯ 5.0 makes asynchronous programming much easier.

Example: Reading from a stream

Synchronous:

```
int bytesRead =
  str.Read(...);
    // read and wait
```

Asynchronous: Task model (2008-)

```
Task<int> task = str.ReadAsync(...); // non-blocking
// do some work
int bytesRead = task.Result; // may block
```

Efficient waiting

Efficient waiting uses callbacks, only invoked once done!

```
Task<int> task = str.ReadAsync(...); // non-blocking

task.ContinueWith(doneTask => { // task done!
  int bytesRead = doneTask.Result; // can't block
  ...
});
```

That’s the easy bit — the tough part is writing the callback.

Sync vs. async taste challenge

C♯ 4.0: Sync vs. Async

Synchronous stream length method

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

C♯ 4.0: Sync vs. Async

Asynchronous stream length method

```
public static Task<long> LengthAsync4(Stream src) {
  var tcs = new TaskCompletionSource<long>();
  var buf = new byte[0x1000]; int bytesRead;
  long totalRead = 0;
  Action<Task<int>> While = null;
  While = rdtask => {
    if ((bytesRead = rdtask.Result) > 0) {
        totalRead += bytesRead;
        src.ReadAsync(buf, 0, buf.Length).ContinueWith(While);
    }
    else tcs.SetResult(totalRead);
  };
  src.ReadAsync(buf, 0, buf.Length).ContinueWith(While);
  return tcs.Task;
}
```

"9/10 devs prefer the taste of synchronous code."

The problem with callbacks

Turning a synchronous call into an asynchronous call is hard! (The "Inversion of control" problem.)

Need to capture the next state of the caller as a callback.

Method state includes locals but also implicit control state:

- program counter (what to do next in this method)
- runtime stack (caller to return to from this method!)

This is easier in languages with call/cc (Scheme/SMLNJ).

Others have to be clever.

(Honorable mentions: Haskell, F#, Scala)

Sync vs. async taste challenge, again

C♯ 5.0: Sync vs. Async

Synchronous stream length method

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

C♯ 5.0 Async version

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

"Mr C♯ compiler: please build the callback for me"

"Wow, almost the same—and it's good for me too?"

Async: Statics

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

C♯ 5.0 adds two keywords `async` and `await`:

- An `async` method must return a `Task<T>` (or `Task` or `void`);
- ...yet exit by `return` of a `T` (not a `Task<T>`).
- Only `async` methods can contain await expressions.
- If `e` has type `Task<U>` then `await e` has type `U`.
- So one async can await task of another.

Async: Dynamics

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

Calling `LengthAsync()`:

- Creates a fresh incomplete task for this call.
- Executes until the next `await` expression.
- If awaited argument is complete now:
  - continue with result (fast path).
  - otherwise, suspend until completion (slow path).
- `return` completes this call's task.
- The first suspend yields the incomplete task to the caller.

Fine points

An async method call:

- runs synchronously until or unless it needs to suspend.
- once suspended, may complete asynchronously.
- if never suspended, completes on same thread.
- does not (in itself) spawn a new thread.

Synchronous on entry avoids context switching; enables fast path implementations.

Typically, suspensions resume in thread pool or event loop; not on new threads.

Threading details depends on context (it's complicated—we don't formalize it, but we could).

What’s the compiler doing?

```
public static Task<long> LengthAsync(Stream src) {
  var tcs = new TaskCompletionSource<long>(); // tcs.Task new & incomplete
  var state = 0; TaskAwaiter<int> readAwaiter;
  byte[] buffer = null; int bytesRead = 0; long totalRead = 0;
  Action act = null; act = () => {
    while (true) switch (state++) {
        case 0: // entry
          buffer = new byte[0x1000]; totalRead = 0; continue; // goto 1
        case 1: // while loop at await
          readAwaiter = src.ReadAsync(buffer, 0, buffer.Length).GetAwaiter();
          if (readAwaiter.IsCompleted) continue; // continue from 2
          else { readAwaiter.OnCompleted(act); return; } // suspend at 2
        case 2: // while loop after await
          if ((bytesRead = readAwaiter.GetResult()) > 0) {
            totalRead += bytesRead;
            state = 1; continue; // goto 1
          }
          else continue; // goto 3
        case 3: // while exit
          tcs.SetResult(totalRead); // complete tcs.Task & "return"
          return; // exit machine
      }
  }; // end of act delegate
  act(); // start the machine on this thread
  return tcs.Task;
} // on first suspend or exit from machine
```

Formal specification

Current C$^\sharp$ 5.0 specs:

- Precise prose describing syntax and typing.
- Example source to source translations (like previous slide).

Hard to follow, trickier to apply.

Our Goal: a precise, high-level operational semantics for:

- programmers (perhaps)
- compiler writers (hopefully)
- researchers (realistically)

No mention of the finite state machine at all!

But for the non-mathematically inclined...

Summary:

- We did the proper "featherweight" thing: $FC^{\sharp}_{5}$
- It works!
  - `async` can be given a "high-level" operational interpretation
  - It can be seen as a "simple" extension of $C^{\sharp}$ 4.0 semantics
  - The correctness proof is quite subtle (as we have shared state)

C$\sharp$ 5.0 operational semantics

Three layered small-step relations:

- Frame transitions (frame local steps):

$$H_1 \triangleright F_1 \rightarrow H_2 \triangleright F_2$$

- Frame stack transitions (thread local steps):

$$H_1 \triangleright FS_1 \twoheadrightarrow H_2 \triangleright FS_2$$

- Process transitions (communication):

$$H_1 \triangleright P_1 \rightsquigarrow H_2 \triangleright P_2$$

where	F	$::=$	$\langle L, \bar{s} \rangle^{\ell}$	Frame (locals plus statements)
	L	$::=$	$\{\overline{x} \mapsto \overline{v}\}$	Locals Map (frame state)
	FS	$::=$	$\epsilon \mid F \circ FS$	Frame Stack (thread)
	ℓ	$::=$	s	synchronous frame label
			$\mathsf{a}(o)$	asynchronous frame for (task) o
	P	$::=$	$\{FS_1, \ldots, FS_n\}$	Process (bag of stacks)

New Stack transitions

Async method call

$$
\begin{array}{ll}
\multicolumn{2}{l}{H_0 \rhd \langle L_0, y_0 \texttt{=} y_1\texttt{.}m\texttt{(}\overline{z}\texttt{);}\overline{s}\rangle^{\ell} \circ FS} \\
\multicolumn{2}{l}{\twoheadrightarrow H_1 \rhd \langle L_1, \overline{t}\rangle^{\mathsf{a}(o)} \circ \langle L_0[y_0 \mapsto o], \overline{s}\rangle^{\ell} \circ FS} \\
\text{where} & H_0(L_0(y_1)) = \langle \sigma_1, FM\rangle \\
 & mbody(\sigma_1, m) = mb\colon (\overline{\sigma}\ \overline{x}) \to^{\mathsf{a}} \psi,\ \text{and } mb = \overline{\tau}\ \overline{y}\texttt{;}\ \overline{t} \\
 & o \notin dom(H_0) \\
 & H_1 = H_0[o \mapsto \langle \psi, \mathsf{running}(\epsilon)\rangle] \\
 & L_1 = [\overline{x} \mapsto L_0(\overline{z}), \overline{y} \mapsto default(\overline{\tau}), \texttt{this} \mapsto L_0(y_1)]
\end{array}
$$

- An `async` call pushes a new active frame (as before).
- new frame is labeled asynchronous $\mathsf{a}(o)$.
- o is address of a fresh, running task (no waiters yet).
- caller will receive task o (once active).

New Stack transitions

Async return

$$
\begin{array}{l}
H_0 \triangleright \{\langle L, \texttt{return}\ y\texttt{;}\overline{s}\rangle^{\mathsf{a}(o)} \circ FS\} \cup P \\
\rightsquigarrow H_1 \triangleright \{FS\} \cup \mathit{resume}(\overline{F}) \cup P \\
\quad \text{where} \quad \begin{array}{l} H_0(o) = \langle \texttt{Task<}\sigma\texttt{>}, \mathsf{running}(\overline{F})\rangle \\ H_1 = H_0[o \mapsto \langle \texttt{Task<}\sigma\texttt{>}, \mathsf{done}(L(y))\rangle] \end{array}
\end{array}
$$

Returning from an asynchronous method atomically:

- Completes the frame's task (retrieved from label!)
- With value "returned".
- Resumes any suspended frames (as threads).

(Label determines meaning of `return` x)

Await transitions

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

- `await` on done task continues with its value.

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

- `await` on running task adds current frame to its waiters.
- Waiters call partial **`GetResult`** which extracts value

Correctness

Usual preservation and progress (modulo usual OO stuck states) technique **but way, WAY, harder**:

- lots of disjointness requirements on task ids in heap, frame stacks, process
- tasks satisfy a protocol:
  $\textsf{running}(\epsilon) \to^* \textsf{running}(F, FS) \to^* \textsf{done}(v)$

(see paper).

Related work

[Too many to mention!]

Highly relevant:

- Millstein et al. TaskJava (PEPM 2007)
  - similar implementation techniques (state machines).
  - no pre-emptive concurrency—just convenient, asynchronous sequential execution.
  - simpler formalization—no heap (!) => no thread communication.
- F♯'s asynchronous workflows:
  - similar ends; different means and tradeoffs.
  - syntactically heavier (do-notation).
  - workflows are (inert) values: easier to compose first, run later.
  - more general (no inherent one-shot restriction).
  - less efficient (every suspend allocates a new continuation).

[Apologies to other Highly relevant authors☺]

Conclusions

- C$\sharp$ 5.0 makes it much easier to write asynchronous code:
  - No need to roll your own callbacks
  - Builds upon existing `Task` library (which has lots of goodies)
- The essence of these new features can be captured precisely:
  - Significant fragment of C$\sharp$ 5.0
  - Builds upon existing formalization
  - No mention of finite-state machine encoding
  - Lots more in the paper:
    1. One-shot semantics
    2. Tail-call optimizations
    3. Awaitable patterns
    4. Exceptions
- Future work: More low-level semantics exposing thread details (including `SynchronizationContext`)

Questions?