The F# Asynchronous Programming Model

Don Syme[1], Tomas Petricek[2], Dmitry Lomov[3]

[1] Microsoft Research, Cambridge, United Kingdom
[2] Faculty of Mathematics and Physics, Charles University in Prague, Czech Republic
[3] Microsoft Corporation, Redmond WA, USA
{dsyme, dmilom}@microsoft.com, tomas@tomasp.net

With one breath, with one flow, you will know, asynchronicity. [The Police, 1983, adapted]

Abstract. We describe the asynchronous programming model in F#, and its applications to reactive, parallel and concurrent programming. The key feature combines a core language with a non-blocking modality to author lightweight asynchronous tasks, where the modality has control flow constructs that are syntactically a superset of the core language and are given an asynchronous semantic interpretation. This allows smooth transitions between synchronous and asynchronous code and eliminates callback-style treatments of inversion of control, without disturbing the foundation of CPU-intensive programming that allows F# to interoperate smoothly and compile efficiently. An adapted version of this approach has recently been announced for a future version of C#.

1 Introduction

Writing applications that react to events is becoming increasingly important. A modern application needs to carry out a rich user interaction, communicate with web services, react to notifications from parallel processes, or participate in cloud computations. The execution of reactive applications is controlled by events. This principle is called *inversion of control* or *the Hollywood principle* ("Don't call us, we'll call you"). Even the internal architecture of multi-core machines is approaching that of an event-based distributed computing environment [2].

For this paper, asynchronous (also called "non-blocking" or "overlapped") programming is characterized by many simultaneously pending reactions to internal or external events. These reactions may or may not be processed in parallel. Today, many practically-oriented languages have reached an "asynchronous programming impasse":

- **OS threads are expensive, while lightweight threading alone is less interoperable.** Despite many efforts to make them cheap, OS threads allocate system resources and large stacks [16] and their use is insufficient for problems that require a large number of pending reactions of outstanding asynchronous communications. For this reason many advocate either complete re-implementations of OS threading [3] or language runtimes supporting only lightweight threads. However, both are difficult without affecting performance of CPU-intensive native code, and in any case interoperating with OS threads is a fundamental requirement for languages that interoperate smoothly with virtual machines such as C#, F#, Scala [12, 19], so in this paper we assume it as an axiom. So these languages must look to add an additional lightweight-tasking model that is not 1:1 with OS threads, a starting point for this paper.

- **Asynchronous programming using callbacks is difficult.** The usual approach to address asynchronous programming is to use callbacks. However, without language support, callback-based code inverts control, is awkward and is limited in expressivity. In normal use, asynchronous programming on .NET and Java leads to a tangle of threads, callbacks, exceptions and data-races.

What is to be done about this? The answer proposed in this paper, and adopted by F# since 2007[1], is to add an *asynchronous modality* as a first-class feature to a general purpose language design. By "modality" we mean reusing the control flow syntax of a host language with a different computational interpretation.[2] The key contribution of this paper is to give a recipe for how to augment a core language (e.g. an ML-like language, with no threading or tasks) with a non-blocking modality to author lightweight asynchronous tasks in a relatively non-intrusive way. The modality has control constructs that are syntactically a superset of the core language and these are given an asynchronous semantic interpretation. For F#, this allows asynchronous code to be described fluently in familiar language syntax, without disturbing the foundation of CPU-intensive programming that allows F# to compile efficiently to Common IL, and hence to native code, and to interoperate well with .NET and C libraries.

2 An Overview of F# Asynchronous Programming

In this section we give an overview of the elements of F# asynchronous programming, element by element. We assume familiarity with ML-like core languages and use *expr* to indicate ordinary expressions in F# programming [17]. The F# asynchronous programming extension adds a new syntactic category *aexpr* to indicate the added syntax of asynchronous expressions:

expr := `async {` *aexpr* `}`

The foundation of F# asynchronous programming is the `Async<T>` type, which represents an asynchronous computation. All expressions of the form `async { ... }` are of type `Async<T>` for some `T`. When executed, an async value will eventually produce a value of type `T` and deliver it to a continuation.

[1] This paper describes the asynchronous support in F# 2.0. While the core idea was released and published in book form 2007, the model has not been described in the conference literature. This paper aims to rectify this and to help enable replication in other languages.

[2] Other examples of language modalities are C# iterators (where the control syntax of C# is used to write programs that generate sequences) and F# sequence expressions (a similar use).

In asynchronous expressions, control-flow constructs can be used to form values that represent asynchronous computations, and additions are made to this syntax to await the completion of other asynchronous computations and bind their results. The grammar of asynchronous expressions for F# is shown below[3]. Importantly, this is a superset of F# core language syntax, where control flow constructs are preferred to have an asynchronous interpretation.

aexpr :=

| `do!` *expr*	execute async
| `let!` *pat* `=` *expr* `in` *aexpr*	execute & bind async
| `let` *pat* `=` *expr* `in` *aexpr*	execute & bind expression
| `return!` *expr*	tailcall to async
| `return` *expr*	return result of async expression
| *aexpr*`;` *aexpr*	sequential composition
| `if` *expr* `then` *aexpr* `else` *aexpr*	conditional on expression
| `match` *expr* `with` *pat* `->` *aexpr*	match expression
| `while` *expr* `do` *aexpr*	asynchronous loop on synchronous guard
| `for` *pat* `in` *expr* `do` *aexpr*	asynchronous loop on synchronous list
| `use` *val* `=` *expr* `in` *aexpr*	execute & bind & dispose expression
| `use!` *val* `=` *expr* `in` *aexpr*	execute & bind & dispose async
| `try` *aexpr* `with` *pat* `->` *aexpr*	asynchronous exception handling
| `try` *aexpr* `finally` *expr*	asynchronous compensation
| *expr*	execute expression for side effects

The signatures of the library functions used in this section are:

```
Async.RunSynchronously    : Async<'T> → 'T
Async.StartImmediate      : Async<unit> → unit
Async.StartInThreadPool[4] : Async<unit> → unit
Async.Parallel            : Async<'T>[] → Async<'T[]>
Async.Sleep               : int →  Async<unit>
```

We also assume a function that takes a URL address and fetches the contents of a web page – we show later in this section how this function is defined.

```
getWebPage : string -> Async<string>
```

2.1 Writing, Composing and Running Asynchronous Computations

Asynchronous computations form a monad and can bind a result from another asynchronous computation using **`let! v = expr in aexpr`**. To return a result, we use the **`return expr`** syntax, which lifts an expression into asynchronous computation. The following example downloads a web page and returns its length:

```
async { let! html = getWebPage "http://www.google.com"
        return html.Length }
```

The expected types are as follows:

`let!` pat_{T} `=` $expr_{\mathrm{Async<T>}}$ `in` $aexpr_{:\ \mathrm{Async<U>}}$	**: Async<U>**
`return` $expr_{\mathrm{T}}$	**: Async<T>**

[3] F# indentation aware syntax allows the omission of the **`in`** keyword.

[4] **`Async.StartInThreadPool`** is called **`Async.Start`** in F# 2.0. We use the former for clarity.

The syntax **`do! expr`** indicates the execution of a subordinate asynchronous operation of type **`Async<unit>`**, the type of an asynchronous operation that does not return a useful result. The following example sleeps 5 sec., resumes, performs a side effect, and sleeps another 5 sec. Note F# is an impure, strict functional language, and, as with other operations in F#, asynchronous computations may have side effects.

```
async { do! Async.Sleep 5000
        printfn "between naps"
        do! Async.Sleep 5000 }
```

The typings for the syntactic elements used here are as follows:

`do!` *expr* `Async<unit>`	**`: Async<unit>`**
aexpr `Async<unit>` ; *aexpr* `Async<T>`	**`: Async<T>`**
expr `unit`	**`: Async<unit>`**

Asynchronous computations can also bind the results of core language expressions using **`let v = `*`expr`*` in `*`aexpr`***, executed using normal expression evaluation:

```
async { let! html = getWebPage "http://www.bing.com"
        let words = html.Split(' ', '\n', '\r')
        printfn "the number of words was %d" words.Length }
```

For the F# version of asynchronous programming, a value of type **`Async<_>`** is best thought of as a “task specification” or “task generator”. Consider this:

```
let sleepThenReturnResult =
    async { printfn "before sleep"
            do! Async.Sleep 5000
            return 1000 }
```

This declaration does not start a task and has no side effects. An **`Async<_>`** must be explicitly run, and side effects will be observed each time it is run. For example, we can explicitly run an asynchronous computation and block for its result as follows:

```
let res = Async.RunSynchronously sleepThenReturnResult
printfn "result = %d" res
```

This runs, as a background operation, a task that prints “before sleep”, then does a non-blocking sleep for 5 sec., and then delivers the result 1000 to the blocking operation. In this case, the function is equivalent to standard blocking code with a pause, but we’ll see a more interesting use in Section 3. The choice to have asyncs be task generators is an interesting one. Alternatives are possible: “hot tasks” that run immediately, i.e. futures, or “cold tasks” that must be started explicitly, but can only be run once. Task-generators are more suitable for a functional language as they eliminate state (e.g. whether a task has been started).

When an asynchronous computation does not produce a result, it can be started as a co-routine, running synchronously until the first point that it yields:

```
let printThenSleepThenPrint =
    async { printfn "before sleep"
            do! Async.Sleep 5000
            printfn "wake up" }

Async.StartImmediate printThenSleepThenPrint
printfn "continuing"
```

This program runs a task that prints “before sleep”, then schedules a callback and prints “continuing”. After 5 sec., the callback is invoked and prints “wake up”.

This raises the question of how the callback is run: is it on a new thread? In a thread pool? Fortunately, .NET has an answer to this. Each running computation in .NET implicitly has access to a *synchronization context*, which for our purposes is a way of taking a function closure and running it “somewhere”. We use this to execute asynchronous callbacks. Contexts feature in the semantics in Section 3.

An asynchronous computation can also be started “in parallel” by scheduling it for execution using the .NET thread pool. The operation is queued and eventually executed through a pool of OS threads using pre-emptive multi-tasking.

```
Async.StartInThreadPool printThenSleepThenPrint
```

2.2 Asynchronous Functions

An asynchronous function is a design idiom where a normal F# function or method returns an asynchronous computation. The typical type signature of an asynchronous function **f** is $\mathbf{ty_1} \rightarrow \ldots \rightarrow \mathbf{ty_n} \rightarrow \mathbf{Async<ty_{return}>}$. For example:

```
let getWebPage (url:string) =
    async { let req = WebRequest.Create url
            let! resp = req.AsyncGetResponse()
            let stream = resp.GetResponseStream()
            let reader = new StreamReader(stream)
            return! reader.AsyncReadToEnd() }
```

This uses additional .NET primitives. It is common that functions are written entirely in this way, i.e. the whole body of the function or method is enclosed in **async { ... }**. (Indeed, in Java/C# versions of an asynchronous language modality, it is natural to support only asynchronous *methods*, and not asynchronous *blocks* or *expressions*).

The above example uses several asynchronous operations provided by the F# library, namely **AsyncGetResponse** and **AsyncReadToEnd**. Both of these are I/O primitives that are typically used at the leaves of asynchronous operations. The key facet of an asynchronous I/O primitive is that it does not block an OS thread while executing, but instead schedules the continuation of the asynchronous computation as a callback in response to an event.[5] Indeed, in the purest version of the mechanism described here, every composite async also has this property: asyncs don’t block at all, not even I/O, except where performing useful CPU computations.

Tail Recursive Functions and Loops. A very common pattern in functional programming is the use of recursive functions. Let’s assume we have a function **receive** of type **unit -> Async<int>** that asynchronously returns an integer, for example by awaiting a message. Now consider an asynchronous function that accumulates a parameter by repeatedly awaiting a message:

[5] The .NET library provides operations through the “Asynchronous Programming Model” (APM) pattern of **BeginFoo**/**EndFoo** methods. The F# library provides **Async.FromBeginEnd** to map these to functions and uses this to wrap primitives to await basic operating signals such as semaphores, to read and write socket connections, and to await database requests.

```
let rec count n =
    async { printfn "count = %d" n
            let! msg = receive()
            return! count (n + msg) }
```

Here, `return!` *expr* is an asynchronous tailcall that yields control to the sub-ordinate async, with finite overall resource usage (neither the stack nor the heap holding continuations grows indefinitely). Note *expr* has type `Async<T>` for some `T`.

Recursive asynchronous functions with asynchronous tailcalls give a very general way to define asynchronous loops. However, the F# and OCaml syntax also allows the direct use of `for` and `while` loops, often combined with the use of imperative data structures such as reference cells. It is useful to extend these to asynchronous code. A variation on a `count` function can be defined as follows:

```
let count =
    async { let n = ref 0
            while true do
                printfn "count = %d" n.Value
                let! msg = receive()
                n := n.Value + msg }
```

2.3 Exception Handling and Resource Compensation

Without a language support, the exception handling in asynchronous computations is extremely difficult [10]. With language support it becomes simple: the `try … with` and `try … finally` constructs can be used in async expressions in the natural way:

```
async { try
            let! primary = getWebPage "http://primary.server.com"
            return primary.Length
        with e ->
            let! backup = getWebPage "http://backup.server.com"
            return backup.Length  }
```

Here, a failure anywhere in the download from the primary server results in the execution of the exception handler and download from the backup server.

Deterministic resource disposal is a language construct that ensures that resources (such as file handles) are disposed at the end of a lexical scope. In F# this is the construct `use val = expr in expr`, translated to `let` ***`val`*** `=` ***`expr`*** `in try` ***`expr`*** `finally` ***`val`***`.Dispose()`. The resource ***`val`*** is freed on exit from the lexical scope.

Resource cleanup in asynchronous code is also difficult without language support [10]. Many OO design patterns for async programming use a "state" object to hold the state elements of a composed asynchronous computation, but this is non-compositional. With language support, state becomes implied by closure, and resource cleanup becomes simple. For example, the `getWebPage` function defined above can be improved as follows:

```
let getWebPage (url:string) =
    async { let req = WebRequest.Create url
            use! resp = req.AsyncGetResponse()
            use stream = resp.GetResponseStream()
            use reader = new StreamReader(stream)
            return! reader.AsyncReadToEnd() }
```

Here the connection, the network stream and reader are closed regardless of whether the asynchronous computation succeeds, fails or is cancelled, even though callbacks and asynchronous responses are implied by the use of the asynchronous syntax.

2.4 Cancellation

A *cancellation mechanism* allows computations to be sent a message to "stop" execution, e.g. "thread abort" in .NET. Cancellation mechanisms are always a difficult topic in imperative programming languages, because compiled, efficient native code often exhibits extremely subtle properties when pre-emptively cancelled at arbitrary machine instructions. However, for asynchronous computations we can assume that primitive asynchronous operations are the norm (e.g. waiting on a network request), and it is reasonable to support reliable cancellation at these operations. Furthermore, it is reasonable to implicitly support cooperative cancellation at specific syntactic points, and additionally through user-defined cancellation checks.

One test of asynchronous programming support in a language is whether cancellation of asynchronous operations is handled without additional plumbing. F# async supports the implicit propagation of a *cancellation token* through the execution of an asynchronous computation. Each cancellation token is derived from a *cancellation capability* (a `CancellationTokenSource` in .NET), used to set the overall cancellation condition. A cancellation token can be given to `Async.RunSynchronously`, `Async.StartImmediate`, `Async.StartInThreadPool` and `Agent.Start`, e.g.

```
let capability = new CancellationTokenSource()
let tasks = Async.Parallel [ getWebPage "http://www.google.com"
                             getWebPage "http://www.bing.com" ]
// Start the work…
Async.Start (tasks, cancellationToken=capability.Token)
// OK, the work is in progress, now cancel it…
capability.Cancel()
```

Cancellation is checked at each I/O primitive, subject to underlying .NET library and O/S support, and before the execution of each `return`, `let!`, `use!`, `try`/`with`, `try`/`finally`, `do!` and `async { ... }` construct, and before each iteration of an asynchronous `while` or `for` loop. For `getWebPage` this means cancellation can occur at several places. But it *cannot* occur during core-language code (e.g. expressions such as library calls, executed for side-effects), and it *cannot* occur in such a way that the resource-reclamation implied by the `use` and `use!` expression constructs is skipped. Cancellation is not necessarily immediately effective: in a multi-core or distributed setting it may take arbitrarily long to propagate the cancellation message.

3 Semantics

We now present a semantics for a simplified version of F# async programming, with the following aims:

- To give a formal reference model that is close to an ideal implementation, yet fairly neutral w.r.t. the core language.

- To differentiate between computations that are *pending on I/O*, *waiting in work queues*, and *actively burning the CPU*. These are the operational characteristics that matter most to working programmers, as they have different cost models.
- To give a semantics that can be reduced to (a) the "single-threaded" model, where one thread serves all reactions, or the "thread-pool" model, where a pool of threads serves all reactions.

We do not present formal proofs based on the given semantics. The semantics is as follows. We first perform a CPS conversion to reduce async expressions to core language expressions (Fig. 1). We assume the "core" language has appropriate contextual reduction rules (see Fig. 2). An async expression becomes a function $\lambda(sc, ec, cc, t).\mathrm{body}$ accepting success, exception and cancellation continuations sc, ec and cc, and a cancellation token t. The only asynchronous action is asyncsleep which raises a wake-up event after an arbitrary time period. Starting an async provides continuations to reify errors and cancellation as exceptions in the core language.

Fig. 2 presents semantics for our asynchronous extension. We assume a core language whose semantics is given as a standard small-step reduction relation $\boldsymbol{e} \rightsquigarrow \boldsymbol{e}'$. The semantics for the asynchronous extension is then specified as a relation on $(A, Q, P) \rightsquigarrow (A', Q', P')$, where

(a) A is a set of active computations $e@ctxt$. Each conceptually corresponds to an active OS thread contending for the CPU, evaluating e. Each is labeled with a synchronization context $ctxt$ indicating how suspended async operations are re-queued. Multiple computations may share the same context (e.g. a thread pool).

(b) Q is a set of queued computations $e@ctxt$. Each conceptually corresponds to a queued work item awaiting execution in $ctxt$. For each context we assume an operation $\mathrm{dequeue}_{ctxt}(Q, A) \rightarrow (Q', A')$ which activates one queued evaluation.

(c) P is a set of pending reactions $ev \rightarrow e@ctxt$. Each conceptually corresponds to a pending callback when ev occurs, e.g. pending reactions to UI events. We assume event descriptors are unique strings indicating a wakeup signal.

REDUCTION performs one step of an active computation in A. SUSPENSION schedules a pending reaction to an event. ACTIVATION activates a queued computation in Q. EVENT queues a pending reaction in response to an event. Evaluation is non-deterministic: more than one reduction rule may apply to a given triple. We do not specify when events are raised: we assume they happen at an arbitrary number of steps once created by evaluations of asyncsleep.

Some important ramifications of the semantics is as follows:

- When there is one *ctxt*, with one thread, the semantics degenerates to a deterministic queue of event reactions, each run to completion or to an asyncsleep.
- When there is one *ctxt*, and multiple threads, the semantics degenerates to a thread pool, running reactions to events in parallel.
- Cancellation cannot be caught, though `finally` clauses are run when cancellation occurs. If an exception happens in a `finally` clause, then if the `finally` is being executed during cancellation, the exception is ignored, otherwise it is propagated.

- Cancellation checks are implicit at specific, well-defined places. Regular non-asynchronous expressions can be used for non-interruptible operations.

$$
\begin{aligned}
\text{async}\,\{\ ae\ \} &\equiv \lambda(sc,ec,cc,t).\,\emptyset;\,\Delta\ ec\ [\![ae]\!]\ (sc,ec,cc,t)\\
\text{asyncsleep} &\equiv \lambda conts.\,\text{suspend}\ (conts, fresh)\\
\text{StartImmediate}(e,t) &\equiv e\ \big((\lambda x.\,x),(\lambda exn.\,\text{raise}\ exn),(\lambda exn.\,\text{raise}\ exn),t\big)\\
[\![\text{let!}\,x = e\ \text{in}\ ae]\!] &= \lambda(sc,ec,cc,t).\,\emptyset;\ e\ \Big(\big(\lambda \text{x}.\,[\![ae]\!]\ (sc,ec,cc,t)\big),ec,cc,\text{t}\Big)\\
[\![\text{return}\ e]\!] &= \lambda(sc,ec,cc,t).\,\emptyset;\ sc\ e\\
[\![\text{if}\ e\ \text{then}\ ae_1\ \text{else}\ ae_2]\!] &= \text{if}\ e_1\ \text{then}\ [\![ae_1]\!]\ \text{else}\ [\![ae_2]\!]\\
[\![\text{let}\ v =\ e\ \text{in}\ ae]\!] &= \text{let}\ v =\ e\ \text{in}\ [\![ae]\!]\\
[\![\text{try}\ ae\ \text{finally}\ e]\!] &\\
&= \lambda(sc,ec,cc,t).\,\emptyset;\Delta\ ec[\![ae]\!]\big((\lambda x.\,\Phi\ e\ sc\ ec\ x),(\lambda x.\,\Phi\ e\ ec\ ec\ x),(\lambda x.\,\text{K}e;\ cc\ x),t\big)\\
[\![\text{try}\ ae_1\ \text{with}\ x \rightarrow\ ae_2]\!] &= \lambda(sc,ec,cc,t).\,\emptyset;\ \Delta\ ec\ [\![ae_1]\!]\ \big(sc,\big(\lambda x.\,[\![ae_2]\!]\ (sc,ec,cc,t)\big),cc,t\big)\\
[\![\text{while}\ e\ \text{do}\ ae]\!] &= \text{let rec f() = if}\ e\ \text{then}\ [\![ae;\ \text{return!}\ \text{f()}]\!]\ \text{else}\ [\![\text{return()}]\!]\ \text{in f()}\\
[\![\text{return!}\ e]\!] &= e\\
[\![e]\!] &= \lambda(sc,ec,cc,t).\,sc\ x\\
\emptyset; e &\equiv \text{if}\ t.\text{Cancelled then}\ c\ \text{"cancel" else}\ e\\
\Phi\ e\ sc\ ec\ x &\equiv \text{match (try Ok}\ e\ \text{with}\ err \rightarrow \text{Err}\ err)\ \text{with Ok}\ v \rightarrow sc\ x \mid \text{Err}\ err \rightarrow ec\ err\\
\Delta\ ec\ f\ x &\equiv \text{match (try Ok}\ (f\ x)\ \text{with}\ err \rightarrow \text{Err}\ err)\ \text{with Ok}\ v \rightarrow v \mid \text{Err}\ err \rightarrow ec\ err\\
\text{K}\ e &\equiv \text{try}\ e\ \text{with}\ err \rightarrow \text{nil}
\end{aligned}
$$

Figure 1. CPS Translation of Asynchronous Expressions[6] [7]

4 Patterns for Concurrent and Reactive Programming

We now present some common patterns built on top of the F# asynchronous model.

4.1 Parallel Composition

Parallel composition of asynchronous computations is efficient because of the scalability properties of the .NET thread pool and the controlled, overlapped execution of operations such as web requests by modern OSs. The F# library provides two simple options for parallel composition, though it is easy to author additional patterns, particularly through the use of agents (see below).

Fork-join parallelism. The library function `Async.Parallel` takes a list of asynchronous computations and creates a single asynchronous computation that starts the individual computations in parallel and waits for their completion:

[6] $\emptyset$ indicates a cancellation check, given a cancellation continuation ***c*** and a cancellation token ***t***. Φ and K indicate detecting and ignoring an exception in core-language code respectively. Δ represents catching an exception and passing it to an exception continuation.

[7] We omit **do!**, **aexpr; aexpr** and **expr** : they are syntactic sugar for **let!**. No cancellation check is inserted for the sub-case **expr; aexpr**. For **match**, **for** and **use** see the F# spec [17]**.**

$e = e\ e \mid \lambda x.e \mid \textbf{let}\ x = e\ \textbf{in}\ e \mid \textbf{raise}\ e \mid \textbf{try}\ e\ \textbf{with}\ v \rightarrow e \mid \textbf{suspend}(e,ev)$ – some core language expressions and results (let rec, tuples, primitive values, conditionals, pattern matching omitted)

$\boldsymbol{A} = \{e@ctxt\}$ – sets of active computations

$\boldsymbol{Q} = \{e@ctxt\}$ – sets of queued computations

$\boldsymbol{P} = \{ev \rightarrow e@ctxt\}$ - sets of pending reactions

$e \rightsquigarrow e'$ – small-step evaluation relation for the core language

Start state is $\boldsymbol{A}_0 = \{\overline{e_\iota @ctxt_\iota}\}$ for some $\overline{e_\iota}$ and $\overline{ctxt_\iota}$, $\boldsymbol{Q_0} = \emptyset$, $\boldsymbol{P_0} = \emptyset$

$\boxed{(\boldsymbol{A}, \boldsymbol{Q}, \boldsymbol{P}) \rightsquigarrow (\boldsymbol{A}', \boldsymbol{Q}', \boldsymbol{P}')}$ evaluation relation for asynchronous extension:

REDUCTION: $e \rightsquigarrow e' \implies (\{e@ctxt\} \cup \boldsymbol{A}, \boldsymbol{Q}, \boldsymbol{P}) \rightsquigarrow (\{e'@ctxt\} \cup \boldsymbol{A}, \boldsymbol{Q}, \boldsymbol{P})$

SUSPENSION: $(\{\textbf{suspend}(e, ev)@ctxt\} \cup \boldsymbol{A}, \boldsymbol{Q}, \boldsymbol{P}) \rightsquigarrow (\boldsymbol{A}, \boldsymbol{Q}, \boldsymbol{P} \cup \{ev \rightarrow e@ctxt\})$

ACTIVATION: $dequeue_{ctxt}(\boldsymbol{Q}, \boldsymbol{A}) \rightarrow (\boldsymbol{Q}', \boldsymbol{A}')$, $ctxt$ is in $\boldsymbol{Q} \implies (\boldsymbol{A}, \boldsymbol{Q}, \boldsymbol{P}) \rightsquigarrow (\boldsymbol{A}', \boldsymbol{Q}', \boldsymbol{P})$

EVENT: $(\boldsymbol{A}, \boldsymbol{Q}, \boldsymbol{P} \cup \{ev \rightarrow e@ctxt\}) \rightsquigarrow (\boldsymbol{A}, \boldsymbol{Q} \cup \{e@ctxt\}, \boldsymbol{P})$

Figure 2. Expression Reduction

```
let task =
    Async.Parallel [ getWebPage "http://www.yahoo.com";
                     getWebPage "http://www.bing.com" ]
let result = Async.RunSynchronously task
```

It is possible to create computations that fetch tens of thousands of web pages in parallel. Assuming that **`urls`** is a list of URLs:

```
let all = Async.Parallel [ for url in urls -> getWebPage url ]
```

Promise-based parallelism. The F# library primitive for parallel execution is **`Async.StartChild`**. Its type is:

```
Async.StartChild  : Async<'T> → Async<Async<'T>>
```

It takes an async representing a child task and returns an async that represents the completion of the task, a form of promise [5]. Two-way parallel composition is then:

```
let parallel2 (job1, job2) =
    async { let! task1 = Async.StartChild job1
            let! task2 = Async.StartChild job2
            let! res1 = task1
            let! res2 = task2
            return (res1, res2)
```

On the first bind, **`StartChild`** starts the computation and returns a promise, also represented as an async, which is awaited on the second bind. The inferred type is:

```
val parallel2 : Async<'T> * Async<'U> -> Async<'T * 'U>
```

4.2 Reactive Agents using State Machines

One primary motivation for including the async modality in F# is that it allows a faithful and simple representation of asynchronous message-receiving agents. An agent encapsulates a message queue and asynchronously reacts to messages received from other components. The signature of the F# library type for agents is as follows:

```
type Agent<'T> =
    static member Start: (Agent<'T> -> Async<unit>) -> Agent<'T>
    member Receive : Async<'T>
    member Post : 'T -> unit
```

(`Agent<T>` is a recommended type alias for the type `MailboxProcessor<T>` in F# 2.0.) One litmus test of an asynchronous programming modality is writing reactive state machines using a set of mutually recursive asynchronous functions. This is a common pattern for reactive agents [20]. For example, consider an agent that adds numbers and can be activated and deactivated. The type of messages sent to the agent is:

```
type Message =
    | Toggle
    | Add of int
    | Get of AsyncReplyChannel<int>
```

The agent has states *active* and *inactive*, which are represented as functions. Both states are parameterized by the current number maintained by the agent. The following example creates and starts the agent (initially *active* with value 0):

```
let agent = Agent<Message>.Start (fun inbox ->
    let rec active n =
        async { printfn "active %d" n
                let! msg = inbox.Receive()
                match msg with
                | Toggle -> return! inactive n
                | Add m  -> return! active (n + m)
                | Get ch -> ch.Reply n; return! active n }
    and inactive n =
        async { printfn "inactive %d" n
                let! msg = inbox.Receive()
                match msg with
                | Toggle -> return! active n
                | Add _  -> return! inactive n
                | Get ch -> ch.Reply n; return! inactive n }
    active 0 )
```

We can use the `Post` member of the agent to send messages to the state machine, e.g.

```
agent.Post (Add 10)   // Prints "active 10"
agent.Post Toggle     // Prints "inactive 10"
agent.Post (Add 20)   // Prints "inactive 10"
```

Results can be retrieved by agents using `PostAndAsyncReply`:

```
async { agent.Post (Add 30)                  // prints: "active 30"
        let! n = agent.PostAndAsyncReply Get // calls & waits
        printfn "got: %d" n }                // prints: "got: 30"
```

4.3 Reactive User Interface Programming

Typical reactive GUI code should not perform CPU intensive calculations, but needs to promptly react to the user activity. This is an area where the F# asynchronous model works well as it enables a co-routine style of programming with a rich set of control constructs [13]. Most of GUI frameworks allow accessing widgets only from a single thread (or do not support threads at all, e.g. JavaScript), making cooperative resumption-based asynchronous tasks are a perfect match for GUI programming.

In F#, user interface events are exposed as values [18] and we can use the **`Async.AwaitObservable`** primitive to use them as asyncs that will resume as soon as an event occurs. For example, assume an event **`wnd.LeftButtonDown`** representing clicks on a window. The following prints information about the first click event:

```
Async.StartImmediate
  async { let! me = Async.AwaitObservable wnd.LeftButtonDown
          printfn "clicke at (%d, %d) in %s" me.X me.Y wnd.Text }
```

The code registers a callback that will be called when the event occurs. The callback is scheduled through the GUI message queue. The example above waits only for the first occurrence of the event. To implement more complex logic, we can use control flow constructs available in the asynchronous modality. For example, consider a computation that reactively loops through three colors, in response to mouse clicks.

```
let semaphoreStates =
  async { while true do
            for light in [green; orange; red] do
              let! _ = Async.AwaitObservable wnd.LeftButtonDown
              wnd.BackgroundColor <- light }

Async.StartImmediate semaphoreStates
```

5 Implementation

At its core, the F# 2.0 implementation of the F# async model is as follows:

- The **`async`** syntax is de-sugared by the compiler as a "computation expression".
- The **`Async<T>`** type is represented as a function that, when run, is given three continuations for success, exceptions and cancellation, and will eventually call one of these. A cancellation token is also supplied as an argument.

Together these perform a localized continuation-passing translation of control-flow and a heap-based allocation of the closures. This is a simple and efficient implementation that also builds on the uniform tailcall support of .NET 4.0. This is in essence a direct implementation of the semantics described in Section 3, though many local optimizations are added, and additional protection is made against some cases where .NET does not guarantee tailcalls, e.g. in some partial-trust execution.

The **`async { ... }`** construct is an instance of an F# *computation expression* [19], a form of *retargetable syntactic control-flow*, c.f. Haskell monadic syntax and LINQ query syntax [11]. We have de-emphasized this here, as adding an asynchronous syntactic modality to a language is independent of its implementation. For example:

```
async { let l = ref []
        for url in urls do
           let! result = getWebPage url
           l := result :: !l
        return !l }
```

is de-sugared to

```
async.Delay(fun () ->
    let l = ref []
    async.Combine(
        async.For(urls, fun url ->
            async.Bind(getWebPage url, fun result ->
                l := result :: l
                async.Zero() )),
        async.Delay(fun () -> async.Return(!l))))
```

5.1 Some Usability and Performance Indicators

The role of F# async is to replace the direct use of OS threads in scalable .NET programming, and to be a "nicer" way of writing the event-based code necessary to achieve true scalability. This is hard to quantify, but one way to see this is to look at the results of a small study [10]. This implements a TCP server using four techniques: C#+OS threads, C#+callback async, F#+OS threads and F# + F# async. Approximate coding time and code lengths were recorded, and the developer was an expert in all areas. This study keeps many variables constant: the VM, GC, OS and underlying library, only the language support changes. The results are below:

	max clients	**C# LoC**	**C# coding**	**F# LoC**	**F# coding**
OS Threads	~1200	~90 lines	~20 mins	~60 lines	~20 mins
Async	> 8000	~330 lines	+ ~3 hours	~60 lines	+ ~10 mins

Comparing scalability and development time for a .NET pseudo-stock quote server [10], .NET 3.5, Dell Optiplex 745, Win 7 Enterprise, 4 GB, 32-bit

The advantages of F# async are clear: > 7x improvement in scalability, and ~18x decrease in time to transition to event/async implementation. This is consistent with the authors' experience of using the mechanism in practice.

The above illustrates the primary benefits of F# async programming against its immediate comparison point on .NET. It is also somewhat useful to compare to other systems implementing agent models. Some comparison points are shown below.

	pingpong 10^5, 1msg	**pingpong 1, 10^7 msg**
F# 2.0 async actors	8.2s/211Mb	5.9s/5.6Mb
Scala 2.8.1 actors	5.5s/166Mb	21.4s/23Mb
Erlang 5.8 processes	(exceeds max agents)	16.8s/6Mb

Agent creation and messaging statistics, Windows 7.
pingpong *n* creates *n* pairs of agent and bounces messages between them. Memory use is steady state private working set. Dell E6400, Intel P9500 2.53Ghz, 2 Core, .NET 4.0, Win7 Enterprise

F# 2.0 per-agent overheads are marginally higher, but message processing is faster. However, a word of caution! *In reality, for all these languages, the in-memory processing costs are nearly always "good enough" for real-world asynchronous programming.* In real-world applications the overheads are often swamped by I/O latencies, I/O waits, graphical rendering or other CPU computations. Further, in client apps, a non-blocking UI can be much more important than reducing CPU usage.

6 Summary

Two major themes run through today's programming landscape: Web and Multi-core. Asynchronous/overlapped/non-blocking network programming is a critical problem for optimizing today's web programming, and compositional, functionally-oriented parallel programming is critical for multi-core programming. The F# async model makes significant practical contributions in both these areas, delivering a clean, efficient and scalable implementation of a compositional asynchronous programming model in the context of a viable applied functional programming language, without disturbing compilation via .NET and interoperability with .NET libraries.

To recap, why is such a modality useful? There are three ways to look at this:

- **Expressivity**: Compositional asynchronous reactions are expressed using sequencing, recursion, pattern matching, conditionals and exception handling. State machines, reactive UIs and agents are simple instantiations of these.
- **Semantic Separation**: Adding an asynchronous modality gives language support to a methodology that separates network I/O and asynchronous message passing from "local" effects such as memory access and console I/O.
- **Scalability**: Event-based programming is still essential to scaling for server-side systems which use OS threads. The performance indicators of Section 5 show how using F# async allows both scaling and efficient coding in this domain.

In practice, the F# asynchronous programming model has consistently proved itself to be an effective tool for multi-core, I/O and agent-programming problems [19, 13, 10].

6.1 Related Work

The topics of parallel, reactive, concurrent and distributed programming have given rise to a vast literature. Some of the key techniques are co-routines, promises, futures and actors [20, 1], synchronous languages [4], functional reactive programming, Join-based thread co-ordination, orchestration languages [22] and light-weight threading, especially Erlang [20]. Task, event, async and fork-join libraries abound, with no language integration. Using monadic delimited continuations for event-based programming is not new [9, 7, 15, 21]. Events v. threads is a major topic in systems research, with papers highlighting the duality of the two approaches, or advocating each [8, 9, 3]. The focus is mostly on systems performance, and less on expressivity.

The F# model ranks as a language integrated implementation of a lightweight task mechanism specifically designed to fluently integrate with high-performance code and interoperate well with existing virtual machines. Others with similar goals include Thorn, the "react" and "continuation" models of Scala and Kilim [6, 14, 16] and the F# model shares much in common with the latter two in the use of a localized CPS transform. This achieves conceptual efficiency by re-utilizing the control syntax of the core language with an asynchronous interpretation.

Acknowledgements. We thank Brian McNamara, Nikolaj Bjorner, Niklas Gustafsson, Simon Peyton Jones, Gregory Neverov, Laurent le Brun, Luke Hoban, Jomo Fisher, Tobias Gedell, and others for their help and advice on the design of the F# async model, and LAMP EPFL for a 2006 sabbatical where this work started.

References

[1] Agha, G.: Actors: a model of concurrent computation in distributed systems. MIT Press, Cambridge, MA, USA (1986)

[2] Baumann, A., et al.: The multikernel: a new OS architecture for scalable multicore systems. In: SOSP '09: Proc. of the ACM SIGOPS 22nd Symp. on OS Principles (2009)

[3] von Behren, R., Condit, J., Brewer, E.: Why events are a bad idea (for high-concurrency servers). In: HOTOS'03: Proc. of the 9th Conf. on Hot Topics in OS (2003)

[4] Berry, G., Gonthier, G.: The ESTEREL synchronous programming language: design, semantics, implementation. Sci. Comput. Program. 19(2), 87–152 (1992)

[5] Friedman, D.P., Wise, D.S.: Aspects of applicative programming for parallel processing. IEEE Trans. Computers 27(4), 289–296 (1978)

[6] Haller, P., Odersky, M.: Scala actors: Unifying thread-based and event-based programming. Theor. Comput. Sci. 410(2-3), 202–220 (2009)

[7] Kiselyov, O.: Delimited control in OCaml, abstractly and concretely. In: Blume, M., Kobayashi, N., Vidal, G. (eds.) FLOPS. LNCS, vol. 6009. Springer (2010)

[8] Lauer, H.C., Needham, R.M.: On the duality of operating system structures. SIGOPS Oper. Syst. Rev. 13(2), 3–19 (1979)

[9] Li, P., Zdancewic, S.: Combining events and threads for scalable network services implementation. SIGPLAN Not. 42(6), 189–199 (2007)

[10] McNamara, B.: F# async on the server side (March 2010), http://tinyurl.com/fsasyncserver, retrieved 5/9/2010

[11] Meijer, E., Beckman, B., Bierman, G.: LINQ: reconciling object, relations and XML in the .NET framework. In: SIGMOD '06: Int. ACM Conf. on Mgmt. of Data. ACM (2006)

[12] Odersky, M., Spoon, L., Venners, B.: Programming in Scala. Artima, USA (2008)

[13] Petricek, T., Skeet, J.: Real World Functional Programming: With Examples in F# and C#. Manning, USA (2009)

[14] Rompf, T., Maier, I., Odersky, M.: Implementing first-class polymorphic delimited continuations by a type-directed selective CPS-transform. In: ICFP '09: Proc. of the 14th ACM SIGPLAN Int. Conf. on Func. Prog. (2009)

[15] Srinivasan, S.: Kilim: A Server Framework with Lightweight Actors, Isolation Types & Zero-copy Messaging. Ph.D. thesis, University of Cambridge (2010)

[16] Srinivasan, S., Mycroft, A.: Kilim: Isolation-typed actors for Java. In: ECOOP '08: 22nd European Conf. on OO Prog. (2008)

[17] Syme, D.: F# 2.0 Language Specification, http://tinyurl.com/fsspec

[18] Syme, D.: Simplicity and compositionality in asynchronous programming through first class events (March 2006), http://tinyurl.com/composingevents, Retrieved: Jan 2010

[19] Syme, D., Granicz, A., Cisternino, A.: Expert F#. Apress (2007)

[20] Virding, R., et al.: Concurrent programming in ERLANG (2nd ed.). Prentice Hall (1996)

[21] Vouillon, J.: OCaml light weight threading library (2002), http://ocsigen.org/lwt/

[22] Wehrman, I., Kitchin, D., Cook, W.R., Misra, J.: A timed semantics of Orc. Theor. Comput. Sci. 402, 234–248 (2008)