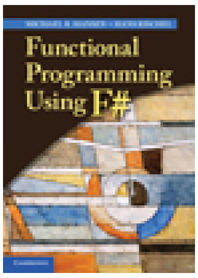

Functional Programming Using F#

Michael R. Hansen, Hans Rischel

Book DOI: http://dx.doi.org/10.1017/CBO9781139093996

Online ISBN: 9781139093996

Hardback ISBN: 9781107019027

Paperback ISBN: 9781107684065

Chapter

13 - Asynchronous and parallel computations pp. 311-338

Chapter DOI: http://dx.doi.org/10.1017/CBO9781139093996.014

Cambridge University Press

13

Asynchronous and parallel computations

This chapter is about programs where the *dynamic* allocation of computer resources like processor time and memory becomes an issue. We consider two different kinds of programs together with programming constructs to obtain the wanted management of computer resources:

1. *Asynchronous, reactive programs* spending most of the wall-clock time awaiting a request or a response from an external agent. A crucial problem for such a program is to minimize the resource demand while the program is waiting.
2. *Parallel programs* exploiting the multi-core processor of the computer by performing different parts of the computation concurrently on different cores.

The construction of asynchronous and parallel programs is based on the hardware features in the computer and software features in system software as described in Sections 13.1 and 13.2. Section 13.3 addresses common challenges and pitfalls in parallel programming. Section 13.4 describes the `async` computation expression and illustrates its use by some simple examples. Section 13.5 describes how asynchronous computations can be used to make reactive, asynchronous programs with a very low resource demand. Section 13.6 describes some of the library functions for parallel programming and their use in achieving computations executing concurrently on several cores.

13.1 Multi-core processors, cache memories and main memory

A typical PC in today's technology (2012) contains two multi-core processor chips, where each processor chip corresponds to Figure 13.1. Programs and data are stored in the main memory while the cache memories contain copies of parts of main memory. Each core gives an independent execution of instructions, and a typical PC offers hence the possibility of four independent executions of instructions. Instructions and data are fetched from the cache memories whenever found there – but have otherwise to be transferred from the main memory. Updating a memory location must always be done in main memory – and in cache if the memory location is cached.

Typical clock frequency of processor is approx. 2 GHz while the clock frequency of main memory is approx. 100 MHz (2012 figures), so cache memory is about 20 times faster than main memory. Maximum speed execution of instructions is hence obtained with instructions and data in cache while the speed may suffer a substantial degradation if there are frequent accesses to main memory. Getting instructions and data in cache may hence give a significant performance gain. Typical memory sizes are 4 GB main memory and 3 MB cache memory

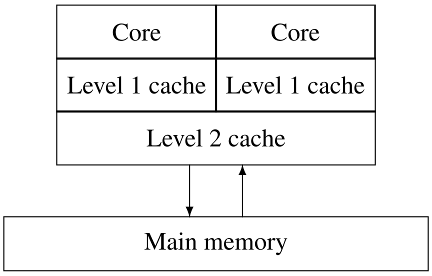

Figure 13.1 A multi-core processor chip in a standard PC

so program and data should fit into the cache unless there is an enormous amount of data – or some other program is using the cache.

The strategies used in managing the cache memories are outside the scope of this book, but one should observe that all program activities on the computer are competing for cache.

13.2 Processes, threads and tasks

This section gives a brief survey of the basic features in operating system and run-time system that are used to manage the concurrent execution of several programs on the computer.

Processes

A *process* is the operating system entity to manage an instance of execution of a free-standing program. The process contains the program and the data of the program execution. A process may comprise multiple threads of execution that execute instructions concurrently. A double-click on an icon on the screen will usually start a process to run the program belonging to the icon.

A free-standing F# program comprises the Common Language Runtime System, CLR (cf. “Interoperating with C and COM” in [13]). The Runtime System manages the memory resources of the process using a stack for each thread and a common heap as described in Chapter 9, and it manages the program execution using threads as described below. A simplified drawing of the memory lay-out of such a process is shown in Figure 13.2.

The `System.Diagnostics.Process` library allows a program to start and manage new processes. This topic is, however, outside the scope of the present book. The reader may consult the Microsoft .NET documentation [9] for further information.

Threads

A *thread* is the .NET vehicle for program execution on one of the cores in the computer. Each thread has its own memory stack and separate execution of instructions. In this chapter we consider only threads managed via a *thread pool* where *tasks* containing programs can be enlisted as *work items*. Such a task will be executed when a thread and a core become available.

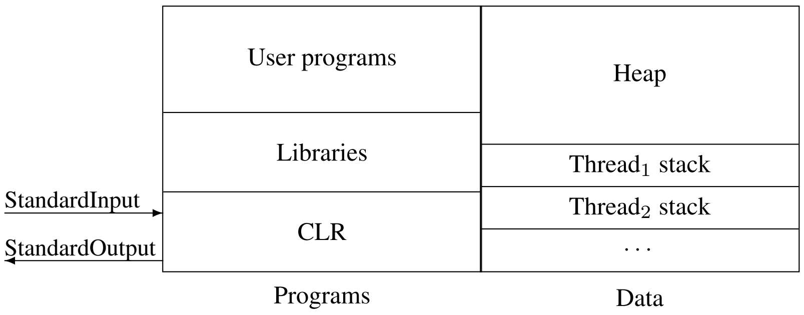

Figure 13.2 Simplified memory lay-out of a process running an F# program

There is a simple example on Page 314 showing creation and start of threads. The reader may consult the description of the `System.Threading` library in [9] for further information.

Tasks

A *task* is a piece of program that can be executed by a thread. When started, a task is *enlisted* as a *work item* in the *thread pool* and it is then executed when a thread becomes available. There are two essentially different ways of executing operations like I/O where the task has to *await the completion* of the operation:

- *Synchronous* operations: The operation is started and the *executing thread* awaits the completion of the operation. The thread continues executing the task when the operation has completed. The standard library I/O functions are synchronous operations.
- *Asynchronous* operations: The operation is started and the task becomes a *wait item* awaiting the completion of the operation. The executing thread is returned to the thread pool to be used by other tasks. The task is again enlisted as a work item when the operation has completed. Asynchronous operations are found in the F# `Async` library and in extensions to the standard I/O libraries.

The continuation of program execution after a *synchronous* operation is done using the stack of the thread where the information is at hand. The mechanism is different in *asynchronous* operations because there is no stack available while the task is waiting. The continuation of the program execution is therefore saved in a special data structure when the asynchronous operations is initiated, and this data structure is then later used to continue the task upon completion of the operation.

A task waiting for completion of an asynchronous operation uses a small amount of memory and no threads and a process may hence run thousands of asynchronous tasks concurrently. The situation is quite different for synchronous tasks where the number is limited by the number of threads available for the program.

These concepts can be illustrated using the "cook-book" metaphor of Section 12.1 where a program is described as a recipe in a cook-book. A process is then described as a restaurant while threads are cooks and tasks are the customer's orders. A synchronous operation corresponds to a cook focussing on the progress of a single order only, while asynchronous operations corresponds to a cook switching between several customer's orders using kitchen stop clocks.

Asynchronous operations are called using `async` computation expressions (cf. Section 13.4).

13.3 Challenges and pitfalls in concurrency

The concurrent execution of tasks gives new challenges and pitfalls for the programmer:

- Management of mutable data shared by several threads.
- Deadlocks caused by threads competing for the same resources.
- Debugging problems where a program failure cannot be reproduced.

Each of these problems is addressed below in a subsection.

Shared mutable data

Updating *mutable* data with a complicated structure may require a number of operations before getting to a well-defined state. The data may hence become garbled if two update operations are intermixed by concurrent execution. This problem is solved by ensuring *mutual exclusion* of updating threads such that two threads do not update the data structure at the same time. The mutual exclusion can be obtained using a `Mutex` object and stipulating the rule that a thread should *acquire* this object before updating the data structure and *release* it after the updating.

The threads `thread1` and `thread2` in the following example both use the mutex object `mutex`. This object is acquired using the `WaitOne` method and released using the `ReleaseMutex` method. The example also illustrates how to create and start threads using F#. The reader may consult the documentation of the `Threading` library in [9] for further details.

```
open System.Threading;;

let mutex = new Mutex();;

let f (n: int) () =
  for k in 1..2 do
    Thread.Sleep 50
    mutex.WaitOne() |> ignore
    printf "Thread %d gets mutex\n" n
    Thread.Sleep 100
    printf "Thread %d releases mutex\n" n
    mutex.ReleaseMutex() |> ignore;;
val f : int -> unit -> unit
```

```
let g() =
  let thread1 = Thread (f 1)
  let thread2 = Thread (f 2)
  thread1.Start()
  thread2.Start();;
val g : unit -> unit

g();;
Thread 2 gets mutex
Thread 2 releases mutex
Thread 1 gets mutex
Thread 1 releases mutex
Thread 2 gets mutex
Thread 2 releases mutex
Thread 1 gets mutex
Thread 1 releases mutex
```

The `System.Collections.Concurrent` library provides thread-safe collections that should be used in place of the corresponding types in the `System.Collections` and `System.Collections.Generic` libraries whenever multiple threads may access the collection concurrently.

These problems of shared mutable data does *not* occur in pure functional programming without mutable data.

Deadlocks

A *deadlock* may occur if two threads $thread_1$ and $thread_2$ are trying to acquire two mutex objects $mutex_1$ and $mutex_2$ simultaneously as follows:

$$\begin{array}{lll} thread_1: & thread_2: & \\ \text{acquire } mutex_1 & \text{acquire } mutex_2 & \\ \text{acquire } mutex_2 & \text{acquire } mutex_1 & \leftarrow \text{ deadlock} \end{array}$$

The deadlock occurs because $thread_1$ is waiting for $mutex_2$ that has been acquired and not yet released by $thread_2$ while $thread_2$ is waiting for $mutex_1$ that has been acquired and not yet released by $thread_1$ – both threads are hence stuck and will never proceed.

A problem of this kind can be solved by stipulating a *fixed order* of nested acquirement of mutex objects to be used throughout the program – for instance always acquire $mutex_1$ before acquiring $mutex_2$ in the example. A thread acquiring $mutex_1$ would then always find a free mutex object $mutex_2$ and can proceed acquiring also $mutex_2$. The thread should eventually release both mutex objects – whereupon another thread may proceed acquiring these objects (in the same order).

A program containing a potential deadlock will work most of the time and the deadlock situation will only occur spuriously in special situations where two threads are doing the reservations at exactly the same time. Extensive testing will most likely *not* reveal the problem and you may end up with an unreliable program that occasionally stops working in stressed situations – exactly when a correct function is most needed.

Debugging problems. Logging facilities

A bug in an asynchronous program may have the unpleasant property that it only leads to failure under special courses of external events and special timing conditions (like the above described deadlock). The traditional scheme of receiving error reports from the customer does *not* work for such programs. The programmer receiving the error report will most likely *not* get a relevant description of the course of events and timing conditions leading to the failure and will be *unable to reproduce the failure* – and hence unable to locate the bug.

The solution to this problem is to include *logging facilities* in the program. These facilities should produce a disk file containing a *log of events* occurring while the program is running. The contents of the log file should be part of the error report from the customer sent in case of a program failure. The log file can then be used by the programmer to trace the course of events leading to the failure – and hopefully to find and correct the bug.

Logging facilities should be a part of any serious real-world asynchronous system. This theme will, however, not be pursued further in this book.

13.4 Asynchronous computations

A value of type `Async<'a>` (for some type `'a`) is an *asynchronous computation*. When started it becomes an executing task that either continues the current task or runs as an independent task. A terminating execution of an asynchronous task of type `Async<'a>` delivers a value of type `'a`.

This concept gives a very elegant notation because an asynchronous computation can be used like any other value as argument or result of a function. Simple asynchronous computations are build using the operations in Table 13.1 inside an `async` expression:

`async {` *asyncExpr* `}`

stream`.AsyncRead: int -> Async<string>`	*stream*`.AsyncRead` n = async.comp. to read n chars from *stream*.
stream`.AsyncRead: byte [] * ?int * ?int -> Async<int>`	*stream*`.AsyncRead(`*buf*`)`, *stream*`.AsyncRead(`*buf*`,`*m*`)` or *stream*`.AsyncRead(`*buf*`,`*m*`,`*n*`)`. Async.comp to read into buf, possibly from pos. m and possibly n chars
stream`.AsyncWrite: string -> Async<unit>`	*stream*`.AsyncWrite` *str* = async.comp. to write *str* to *stream*
stream`.AsyncWrite: string * int * int -> Async<unit>`	*stream*`.AsyncWrite(`*str*`,`*m*`,`*n*`)` = asc.cmp. to write *n* chars of *str* from pos. *m*
webClient`.Async.DownloadString: Uri -> Async<string>`	async.comp. to get WEB source determined by *webClient*.
webRequest`.AsyncGetResponse: unit -> Async<WebResponse>`	async.comp. to await response on web-request
`Async.Sleep: int -> Async<unit>`	`Async.Sleep` n = async.comp. sleeping n mS

A question mark (?) signals an optional argument.

Table 13.1 *Selected asynchronous operations*

`Async.Parallel: Seq<Async<'a>> -> Async<'a []>`	
`Async.Parallel [`c_0`; ...;`c_{n-1}`]`	= async.comp. of $c_0, \ldots, c_{n-1}$ put in parallel

Table 13.2 *Function combining asynchronous computations*

Function	Description
`Async.RunSynchronously: Async<'a> * ?int * ?CancellationToken -> 'a`	Activates async.comp. possibly with time-out in mS and possibly with specified cancellation token. Awaits completion.
`Async.Start: Async<unit> * ?CancellationToken -> unit`	Activates async.comp. possibly with specified canc. token. Does not await completion.
`Async.StartChild: Async<'T> * ?int -> Async<Async<'T>>`	Activates async.comp. and gets async.comp. to await result
`Async.StartWithContinuations: Async<'T> * ('T -> unit) * (exn -> unit) * (OperationCanceledException -> unit) * ?CancellationToken -> unit`	Activates async.comp. with specified continuation and possibly specified cancellation token. Does not await completion of computation.
`Async.FromContinuations: (('T -> unit) * (exn -> unit) * (OperationCanceledException -> unit) -> unit) -> Async<'T>`	Makes current asynchronous task a wait item. Argument function is called with triple of trigger functions as the argument and should save one or more of these closures in variables. The task continues as a work item when a trigger function is called.

Table 13.3 *Selected functions to activate or deactivate asynchronous computations*

The function `Async.Parallel` in Table 13.2 is used to put asynchronous computations in parallel. The computations are started using the functions in Table 13.3. These tables do only contain a selection of functions – further information can be found in [9].

Simple examples of asynchronous computations

Using the asynchronous operation *webClient*`.AsyncDownloadString` we may build an asynchronous computation to download the HTML-source of the DTU home page:

```
open System ;; open System.Net;;  // Uri, WebClient
let downLoadDTUcomp =
  async {let webCl = new WebClient()
         let! html =
           webCl.AsyncDownloadString(Uri "http://www.dtu.dk")
         return html} ;;
val downLoadDTUcomp : Async<string>
```

This is just a value like any other – but if started, it will run a task to download the HTML-source of the DTU home page. This task will do the following:

1. Create a `WebClient` object. This declaration need actually not be part of the `async` expression and could hence be placed before `async {...}`
2. Initiate the download using `AsyncDownloadString`. This function makes the task an wait item and returns this item in the form of an `Async` value *comp*. The asynchronous computation *comp* will eventually run and terminate when the download has completed.
3. The termination of *comp* re-starts the rest of the computation with the identifier `html` bound to the result of *comp* (which in this case is the result of the download).
4. The expression `return html` returns the value bound to `html`, that is, the result of the download.

Please observe the following

- The computation uses very few resources while waiting for the download – it uses for instance no thread during this time period.
- The `let!` construct is required to make a binding to a value that is later returned at the termination of an asynchronous computation.
- The computation expression does in most cases contain a construct like `return` or `return!` to give a result – and will otherwise give the dummy value "`()`" as the result. Using `return!` yields a new asynchronous computation.

Functions computing asynchronous computations

We may generalize the above example to a function computing the asynchronous download computation for arbitrary URL:

```
let downloadComp url =
    let webCl = new WebClient()
    async {let! html = webCl.AsyncDownloadString(Uri url)
           return html};;
val downloadComp : string -> Async<string>
```

Computations downloading the HTML sources of the DTU and Microsoft home pages may then be obtained as function values:

```
let downloadDTUcomp = downloadComp "http://www.dtu.dk";;
val downloadDTUcomp : Async<string>

let downloadMScomp = downloadComp "http://www.microsoft.com";;
val downloadMScomp : Async<string>
```

A computation downloading the HTML-sources corresponding to an array of URL's in parallel can be made using `Async.Parallel` and `Array.map`:

```
let downlArrayComp (urlArr: string[]) =
    Async.Parallel (Array.map downloadComp urlArr);;
val downlArrayComp : string [] -> Async<string []>
```

and we may hence download the HTML-sources of the DTU and the Microsoft home pages concurrently and compute their lengths:

```
let paralDTUandMScomp =
    downlArrayComp
        [|"http://www.dtu.dk"; "http://www.microsoft.com"|];;
val paralDTUandMScomp : Async<string []>

Array.map (fun (s:string) -> s.Length)
          (Async.RunSynchronously paralDTUandMScomp);;
val it : int [] = [|45199; 1020|]
```

The parallel download of HTML-sources can instead be made using the `StartChild` function. This gives separated activation and waiting for completion of two child tasks:

```
let parallelChildrenDTUandMS =
    async {let! compl1 = Async.StartChild downloadDTUcomp
           let! compl2 = Async.StartChild downloadMScomp
           let! html1 = compl1
           let! html2 = compl2
           return (html1,html2)};;
val parallelChildrenDTUandMS : Async<string * string>
```

The calls of `StartChild`:

```
let! compl1 = Async.StartChild downloadDTUcomp
let! compl2 = Async.StartChild downloadMScomp
```

start the downloads in two child tasks in parallel. The identifiers `compl1` and `compl2` are bound to two *asynchronous computations* that *when started* will await the completion of the child tasks. The main task is hence *not* blocked by the `StartChild` operations. It becomes blocked when `compl1` is started and awaits the completion of the corresponding child task:

```
let! html1 = compl1
```

The next `let!` construct: `let! html2 = compl2` will in the same way afterwards await the completion of the second child task.

Exception and cancellation

Executing `async` computations includes handling *premature termination* caused by an *exception* or a *cancellation*. These concepts can be informally explained using the "cook-book recipe" metaphor of Section 12.1 by imaging a cook working in a restaurant. The cook should not only follow the recipe when processing a customer's order but also handle the following abnormal situations:

- *Exception:* An exception has occurred (like break-down of an oven).
- *Cancellation:* The customer has cancelled the order.

A task executing an `async` computation reacts in case of an *exception* or a *cancellation* by calling the corresponding *continuation*. A cancellation is requested (from outside the task) by setting the *cancellation token* of the execution of the computation (see example below). The cancellation token is polled regularly by the asynchronous library functions and by the member functions of the `async` computation expression. The cancellation is performed with a proper clean-up of resources as soon as the cancellation request has been discovered.

Using the library function `Async.StartWithContinuations` you may supply your own continuations when an asynchronous computation is started. This function requires three continuations among its parameters:

- Normal continuation *okCon* – invoked after normal termination.
- Exception continuation *exnCon* – invoked if an exception is raised.
- Cancellation continuation *canCon* – invoked if the computation is cancelled.

The following example executes the above function `downloadComp` with continuations:

```
open System.Threading;;       // CancellationTokenSource

let okCon (s: string) = printf "Length = %d\n" (s.Length);;
let exnCon _ = printf "Exception raised\n";;
let canCon _ = printf "Operation cancelled\n";;

let downloadWithConts url =
    use ts = new CancellationTokenSource()
    Async.StartWithContinuations
        ((downloadComp url),okCon,exnCon,canCon,ts.Token)
    ts;;
val downloadWithConts : string -> CancellationTokenSource
```

A computation started by a call of `downloadWithConts` may terminate normally:

```
downloadWithConts "http://www.microsoft.com" |> ignore;;
val it : unit = ()
Length = 1020
```

it may be terminated by an exception:

```
downloadWithConts "ppp" |> ignore;;
Exception raised
val it : unit = ()
```

or it may be cancelled:

```
let ts = downloadWithConts "http://www.dtu.dk";;
ts.Cancel();;
val it : unit = ()
Operation cancelled
```

Note the following:

- The task started by `Async.StartWithContinuations` *terminates* when the selected continuation returns the dummy value "`()`". A meaningful program would hence use continuations that initiate some other activity – for instance by sending a message to a queue or activating another task.
- Each *execution* of an asynchronous computation with possible cancellation should have a *fresh* cancellation token source. The above function `downloadWithConts` ensures that by including the allocation in the function declaration.
- Requesting cancellation using `Cancel` on the token source can be followed by a call of one of the *other* continuations if an error occurs or if the operation terminates before the cancellation gets through.

13.5 Reactive programs

A reactive program performs operations using asynchronous waiting and may hence be used to perform many long lasting I/O operations simultaneously while also communicating with the user at the same time. Such a program can be implemented using an *asynchronous event queue*. It is a queue containing events of the following kinds, for example:

- Mouse clicks or key-strokes or other kinds of user input.
- Responses from asynchronous operations.

Asynchronous event queue

An asynchronous event queue from the class `AsyncEventQueue` supports two operations:

```
ev.Post    : 'T -> unit
ev.Receive : unit -> Async<'T>
```

where

- *ev*`.Post` *msg*: inserts the element *msg* in the event queue *ev*.
- *ev*`.Receive()`: Awaits the next element in the event queue *ev*.

The event queue class `AsyncEventQueue` is kindly provided by Don Syme, and its implementation is shown in Table 13.4. It is possible that this queue will be included in the F# standard library.

Design of dialogue programs

We shall now consider the design of reactive programs where the system may engage in a dialogue with a user. The systems considered here will have to react to two kinds of events:

- Input from a user.
- Status events from asynchronous computations.

These events are handled by use of an asynchronous event queue.

```
// An asynchronous event queue kindly provided by Don Syme
type AsyncEventQueue<'T>() =
    let mutable cont = None
    let queue = System.Collections.Generic.Queue<'T>()
    let tryTrigger() =
        match queue.Count, cont with
        | _, None -> ()
        | 0, _ -> ()
        | _, Some d ->
            cont <- None
            d (queue.Dequeue())

    let tryListen(d) =
        if cont.IsSome then invalidOp "multicast not allowed"
        cont <- Some d
        tryTrigger()

    member x.Post msg = queue.Enqueue msg; tryTrigger()
    member x.Receive() =
        Async.FromContinuations (fun (cont,econt,ccont) ->
            tryListen cont)
```

Table 13.4 *An implementation of* `AsyncEventQueue` *by Don Syme*

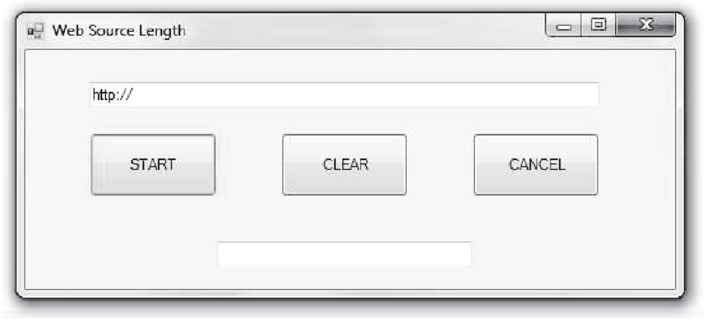

Figure 13.3 Window of asynchronous dialogue program

Consider for example a primitive dialogue program that finds lengths of HTML-sources of web-pages. The program shows a window as in Figure 13.3. The upper text box is used to enter the URL while the lower text box shows the answer from the program. The buttons have the following functions:

`Start` *url*`:` Starts the download of the web-page using the URL in the upper text box.
`Clear:` Clears the text boxes.
`Cancel:` Cancels a progressing download.

The program we shall construct make use of the asynchronous event queue shown in Table 13.4 and it has three parts:

```
open System
open System.Net
open System.Threading
open System.Windows.Forms
open System.Drawing

// The window part
let window =
    new Form(Text="Web Source Length", Size=Size(525,225))
let urlBox =
    new TextBox(Location=Point(50,25),Size=Size(400,25))
let ansBox = new TextBox(Location=Point(150,150),Size=Size(200,25))
let startButton =
    new Button(Location=Point(50,65),MinimumSize=Size(100,50),
               MaximumSize=Size(100,50),Text="START")
let clearButton =
     new Button(Location=Point(200,65),MinimumSize=Size(100,50),
                MaximumSize=Size(100,50),Text="CLEAR")
let cancelButton =
    new Button(Location=Point(350,65),MinimumSize=Size(100,50),
               MaximumSize=Size(100,50),Text="CANCEL")
let disable bs = for b in [startButton;clearButton;cancelButton] do
                     b.Enabled  <- true
                 for (b:Button) in bs do
                     b.Enabled  <- false

// The dialogue part from Table 13.7 belongs here

// Initialization
window.Controls.Add urlBox
window.Controls.Add ansBox
window.Controls.Add startButton
window.Controls.Add clearButton
window.Controls.Add cancelButton
startButton.Click.Add (fun _ -> ev.Post (Start urlBox.Text))
clearButton.Click.Add (fun _ -> ev.Post Clear)
cancelButton.Click.Add (fun _ -> ev.Post Cancel)
// Start
Async.StartImmediate (ready())
window.Show()
```

Table 13.5 *Dialogue program: The Window, Initialization and Start parts*

- The first part contains declarations corresponding to the window shown in Figure 13.3. These declarations are shown in Table 13.5. In this part buttons and text boxes are declared. Furthermore, a function `disable` is declared that controls enable/disable of the buttons in the window. During the download of a web-page, for example, the user should have the option to cancel the ongoing download; but the buttons for clearing the text fields and for starting up a new download should be disabled in that situation.

- The second part (see comments in Table 13.5) contains the dialogue program. We shall focus on this part in the following.
- The third part connects the buttons of the user interface to events, shows the window and starts the dialogue program. This part is shown in the lower part of Table 13.5.

Notice that the program is an event-driven program with asynchronous operations all running on a single thread. The complete program is found at the homepage for the book.

Dialogue automaton

We shall design an event-driven program that reacts to user events and status events from asynchronous operations. The user events are described above. An asynchronous download of a web-page can result in three kinds of status events:

`Web` *html*: The event containing the html-source of a web-page.
`Cancelled`: The event signalling a successful cancelling of a download.
`Error`: The event signalling an unsuccessful download of a web-page possibly due to an illegal URL.

The system must perform some *actions* in response to incoming events, for instance: the action corresponding to a `Clear` event is that the text boxes are cleared, the action corresponding to a `Start` *url* event is the start of an asynchronous download of the web-page *url*, and the action corresponding to a `Web` *html* event prints the number of characters in the string *html* in the lower text box.

The possible sequences of events of an reactive program are often conveniently described by a simple automaton. An *automaton* is a directed graph, with a finite number of *vertices* also called called *states* and *edges* also called *transitions*. A specific state is called the *initial state*. A transition is labelled with a set of events. A *path* of the automaton is a sequence:

$$path = s_0 \xrightarrow{e_1} s_1 \xrightarrow{e_2} \cdots \xrightarrow{e_n} s_n$$

where there is a transition labelled e_i from s_{i-1} to s_i, for $1 < i \leq n$. The sequence of events $e_1\, e_2 \cdots e_n$ is called a *run*.

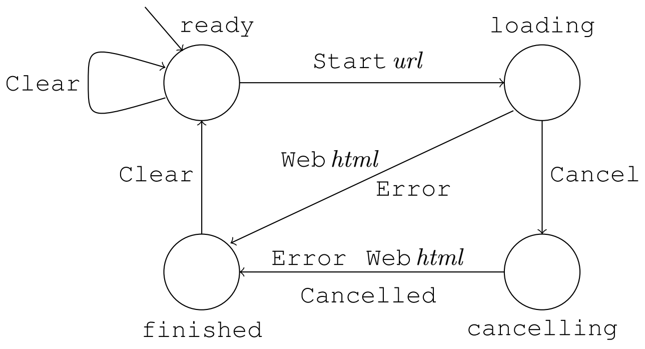

Figure 13.4 A simple dialogue automaton

Consider the automaton in Figure 13.4 with states: `ready`, `loading`, `cancelling` and `finished`, where `ready` is the initial state – it is marked with an incoming arrow, and six events: `Start` *url*, `Clear`, `Cancel`, `Web` *html*, `Cancelled` and `Error`.

The runs starting in the initial state `ready` describe the allowed sequences of events. For example, the sequence

$$\texttt{Clear Start}(url_1)\ \texttt{Web}(html_1)\ \texttt{Clear Start}(url_2)$$

is allowed because it "brings the automaton" from the `ready` state to the `loading` state. Other allowed sequences are:

$$\begin{array}{l}\texttt{Clear Start}(url_1)\ \texttt{Web}(html_1)\ \texttt{Clear Start}(url_2)\\ \texttt{Start}(url_1)\ \texttt{Cancel Cancelled Clear}\\ \texttt{Start}(url_1)\ \texttt{Cancel Web}(html_1)\end{array}$$

while the following sequence is forbidden:

$$\texttt{Start}(url_1)\,\texttt{Cancel Clear}$$

because the automaton gets stuck in the `cancelling` state. The two first events:

$$\texttt{Start}(url_1)\ \texttt{Cancel}$$

lead to the `cancelling` state and there are no outgoing transition labelled `Clear` from that state.

Notice that the automaton conveys an overview of the interaction in the system. Furthermore, the corresponding dialogue program will systematically be constructed from the dialogue automaton in terms of four mutually recursive functions corresponding to the four states. This leads to the program skeleton in Table 13.6.

The only parts that are missing in this program skeleton relate to the actions for the incoming events of the states. The other parts are systematically derived from the dialogue automaton in Figure 13.4. The function implementing a given state of the automaton, for example, the `ready` state, has three parts:

Part 1: Actions corresponding to the incoming event are performed in the first part. This part is not described in the skeleton because these details are not present in the automaton.

Part 2: Forbidden user input is disabled. By inspection of the events labelling the transitions leaving a state, it can be observed which input the user should be able to provide in that state. The `ready` state has no outgoing transition labelled `Cancel` and the corresponding button is therefore disabled.

Part 3: Wait for incoming events and make a corresponding state transition. In the `ready` state only `Start` and `Clear` events are allowed. A `Clear` event leads back to the `ready` state while a `Start` event leads to the `loading` state.

The program skeleton in Table 13.6 contains a few details not present in Figure 13.4: Answer strings are passed from the `loading` and `cancelling` states to the `finished` state.

```
let ready() =
    async { .... // actionReady: actions for incoming events
           disable [cancelButton]
           let! msg = ev.Receive()
           match msg with
           | Start url -> return! loading(url)
           | Clear     -> return! ready()
           | _         -> failwith("ready: unexpected message")}

and loading(url)  =
    async { .... // actionLoading: actions for incoming events
           disable [startButton; clearButton]
           let! msg = ev.Receive()
           match msg with
           | Web html ->
             let ans = "Length = " + String.Format("0:D",html.Length)
             return! finished(ans)
           | Error    -> return! finished("Error")
           | Cancel   -> ts.Cancel()
                         return! cancelling()
           | _        -> failwith("loading: unexpected message")}

and cancelling() =
    async { .... // actionCancelling: actions for incoming events
           disable [startButton; clearButton; cancelButton]
           let! msg = ev.Receive()
           match msg with
           | Cancelled | Error
           | Web _ -> return! finished("Cancelled")
           | _    ->  failwith("cancelling: unexpected message")}

and finished(s) =
    async { .... // actionFinished: actions for incoming events
           disable [startButton; cancelButton]
           let! msg = ev.Receive()
           match msg with
           | Clear -> return! ready()
           | _     ->  failwith("finished: unexpected message")}
```

Table 13.6 *Skeleton program for automaton in Figure 13.4*

It is now straightforward to complete the whole dialogue program. The type for events (or messages) and an event queue are declared as follows:

```
type Message = | Start of string | Clear | Cancel
               | Web of string | Error | Cancelled;;

let ev = AsyncEventQueue();;
```

and the action parts missing in the skeleton program are declared as follows:

- Actions for incoming event in the ready state: The two text boxes must be cleared:

```
urlBox.Text <- "http://"
ansBox.Text <- ""
```

- Actions for incoming event in the loading state: The text box for the answer is set and an asynchronous download of a web-page is started with continuations as we have seen on Page 320:

```
ansBox.Text <- "Downloading"
use ts = new CancellationTokenSource()

Async.StartWithContinuations
    (async { let webCl = new WebClient()
             let! html = webCl.AsyncDownloadString(Uri url)
             return html },
      (fun html -> ev.Post (Web html)),
      (fun _ -> ev.Post Error),
      (fun _ -> ev.Post Cancelled),
      ts.Token)
```

- Actions for incoming event in the cancelling state: The answer text box is set.

```
ansBox.Text <- "Cancelling"
```

- Actions for incoming event in the finished state: The answer text box is set.

```
ansBox.Text <- s
```

The complete program for the dialogue automaton is found in Appendix C.

A summary of the approach

We have considered the design of reactive systems in terms of a very simple example in order to be able to focus on the principle elements of the approach. Many systems have a similar form where a system is engaging in a dialogue with users and external sources like database servers on the basis of asynchronous communication.

The type `Message` in the example, provide an abstract notion of the important events in the system that abstracts away the concrete interactions with the user interface. A dialogue automaton provides a convenient technique to define the legal sequences of events in the system. This automaton conveys the essence of the dialogue design in a succinct manner and a dialogue program can systematically be derived from this automaton.

The technical advantage of the approach is that the resulting asynchronous program is executed in a single thread requiring limited computational resources.

13.6 Parallel computations

The Task Parallel Library of the .NET platform provides a powerful framework for exploiting multi-core parallelism. In this section we shall show that functional programming provides an adequate platform for a programmer wanting to exploit this parallelism to speedup the programs. Obtaining a parallel implementation of a side-effect free program makes the correctness problem of the parallel version simple and good library support for parallelism makes the step to the parallel version manageable.

We shall distinguish between two kinds of parallelism: *data parallelism*, where the same function is applied in parallel on distributed data, and *task parallelism*, where a complex problem is solved by combining solutions to simpler problems, that can be solved in parallel. When measuring the effect of multiple cores we consider "big" problems in terms of computation requirements. It does not pay off to parallelize small problems due to the management overhead needed for multiple cores.

In order to experiment with parallelization, we need primitive operations that demands some computation resources in order to make the effect of parallelization visible. Throughout this section a prime number test on randomly generated integers will be used for that purpose. The prime-number test is performed by the function:

```
let isPrime =
    let rec testDiv a b c =
        a>b || c%a <> 0 && testDiv (a+1) b c
    function
    | 0 | 1 -> false
    | n     -> testDiv 2 (n-1) n;;
val isPrime : int -> bool
```

The locally declared function `testDiv` $a\ b\ c$ is true if no integer i where $a \leq i \leq b$ divides c. Testing whether n, with $n > 1$, is a prime number, it suffices to test whether no integer between 2 to $\sqrt{n}$ divides n. In order to use more computing resources `isPrime` is inefficiently implemented by performing this test from 2 to $n - 1$.

```
isPrime 51;;
val it : bool = false

isPrime 232012709;;
val it : bool = true
```

A test of a small number is fast whereas a test of a large prime number like the one in the above example takes some observable amount of time.

We shall use randomly generated integers in our experiments. They are generated by the following function `gen`, where `gen` *range*, with *range* > 0, generates a number that is greater than or equal to 0 and smaller than *range*:

```
let gen = let generator = new System.Random()
          generator.Next;;
val gen : int -> int
gen 100;;
val it : int = 24
```

```
gen 100;;
val it : int = 53
```

The experiments in the rest of this section are conducted on a 4-core 2.67 GHz Intel I7 CPU with 8GB shared memory.

Data parallelism

The map function on collections is the canonical example for exploiting data parallelism, where a function is applied in parallel to the members of a collection. Parallel implementations of functions on arrays are found in the `Array.Parallel` library as shown in Table 13.7.

```
choose    : ('T -> 'U option) -> 'T [] -> 'U []
collect   : ('T -> 'U []) -> 'T [] -> 'U []
init      : int -> (int -> 'T) -> 'T []
iter      : ('T -> unit) -> 'T [] -> unit
iteri     : (int -> 'T -> unit) -> 'T [] -> unit
map       : ('T -> 'U) -> 'T [] -> 'U []
mapi      : (int -> 'T -> 'U) -> 'T [] -> 'U []
partition : ('T -> bool) -> 'T [] -> 'T [] * 'T []
```

Table 13.7 *Functions in the library:* `Array.Parallel`

We have studied these functions previously in the book, so we just illustrate the advantage of using the function parallel version of the `map` function on an array with 5000000 numbers:

```
let bigArray = Array.init 5000000 (fun _ -> gen 10000);;
val bigArray : int [] = [|2436; 7975; 2647; 1590; 5959; 3951;
                          430; 1705; 2527; 1004; 2333; ... |]
```

Mapping the `isPrime` function on the elements of `bigArray` will generate a new Boolean array, where an entry is true if and only if the corresponding entry in `bigArray` is a prime number:

```
#time;;

Array.Parallel.map isPrime bigArray;;
Real: 00:00:05.292, CPU: 00:00:20.592,
GC gen0: 0, gen1: 0, gen2: 0
val it : bool [] = [|false; false; true; false; false; false;
                     false; false; false; false; true; ...|]

Array.map isPrime bigArray;;
Real: 00:00:10.220, CPU: 00:00:10.218,
GC gen0: 0, gen1: 0, gen2: 0
val it : bool [] = [|false; false; true; false; false; false;
                     false; false; false; false; true; ...|]
```

The experiment shows a speed-up of approximately 2 in real time when using the parallel version of map. The main point is that achieving this speed-up is effortless for the programmer. Note that the total CPU time (20.218 seconds) used on all cores is approximately double the time needed for a non-parallel version.

In order to use the library for parallel operations on sequences you need to install the F# Power Pack. The `PSeq` library in that package provides parallel versions of a rich collection of the functions in the `Seq` library (see Chapter 11). These functions can also be used on lists and arrays as we have seen in Section 11.7. We just show one experiment with the `exists` function from the `PSeq` library:

```
#r @"FSHarp.PowerPack.Parallel.Seq"
open Microsoft.FSharp.Collections

let bigSequence = Seq.init  5000000 (fun _ -> gen 10000);;
Real: 00:00:00.001, CPU: 00:00:00.000,
GC gen0: 0, gen1: 0, gen2: 0
val bigSequence : seq<int>

Seq.exists (fun i -> isPrime i && i>10000) bigSequence;;
Real: 00:00:11.557, CPU: 00:00:11.528,
GC gen0: 247, gen1: 3, gen2: 1
val it : bool = false

PSeq.exists (fun i -> isPrime i && i>10000) bigSequence;;
Real: 00:00:05.985, CPU: 00:00:22.183,
GC gen0: 250, gen1: 1, gen2: 0
val it : bool = false
```

In the example we search for the existence of a prime number that do not exists in the generated sequence in order to be sure that the whole sequence is traversed. The speed-up is about 2 and the figures are similar to those for the above experiment using `map`.

Task parallelism

The problem-solving strategy we have used throughout the book is to solve a complex problem by combining solutions to simpler problems. This strategy, which also is known as *divide and conquer*, fits very well with task parallelism, where a complex problem is solved by combining solutions to simpler problems that can be solved in parallel.

We illustrate the idea on a simple example. Consider the type for binary trees given in Section 6.4:

```
type BinTree<'a> = | Leaf
                   | Node of BinTree<'a> * 'a * BinTree<'a>;;
```

A function to test for the existence of an element in a binary tree satisfying a given predicate is declared as follows:

```
let rec exists p t =
    match t with
    | Leaf                 -> false
    | Node(_,v,_) when p v -> true
    | Node(tl,_,tr)        -> exists p tl || exists p tr;;
val exists: ('a -> bool) -> BinTree<'a> -> bool
```

The divide and conquer strategy is employed in the last clause: In order to check whether an element in a tree satisfies the given predicate, the check is performed in the left and right subtrees and those results are combined.

We shall generate trees using the following function:

```
let rec genTree n range =
    if n=0 then Leaf
    else let tl = genTree (n-1) range
         let tr = genTree (n-1) range
         Node(tl, gen range, tr);;
val genTree : int -> int -> BinTree<int>

let t = genTree 25 10000;;
```

The value of `genTree`n *range* is a balanced binary tree with depth n, where every element v occurring in a node is an integer satisfying $0 \leq v < \mathit{range}$. The generated tree `t`, therefore, has 2^{25} leaves and searching through the whole tree is a time-consuming operation:

```
exists (fun n -> isPrime n && n>10000) t;;
Real: 00:01:22.818, CPU: 00:01:22.727,
GC gen0: 0, gen1: 0, gen2: 0
val it : bool = false
```

The obvious idea for parallelization is to do the search in the left and right subtrees in parallel and combine their results. The function `Task.Factory.StartNew` from the namespace `System.Threading.Tasks` is used to create and start a new task:

```
open System.Threading.Tasks;;
let rec parExists p t =
    match t with
    | Leaf                 -> false
    | Node(_,v,_) when p v -> true
    | Node(tl,_,tr)        ->
      let b1 = Task.Factory.StartNew(fun () -> parExists p tl)
      let b2 = Task.Factory.StartNew(fun () -> parExists p tr)
      b1.Result||b2.Result;;
val parExists: ('a -> bool) -> BinTree<'a> -> bool
```

Evaluation of the declaration

```
let b1 = Task.Factory.StartNew(fun () -> parExists p tl)
```

will create and start a task object of type `Task<bool>` and `b1` is bound to that object (similarly for the declaration of `b2`). The property `Result` gets the result of the task upon its completion.

This parallel version does, however, not give any significant performance gain:

```
parExists (fun n -> isPrime n && n>10000) t;;
Real: 00:01:19.659, CPU: 00:04:43.578,
GC gen0: 2972, gen1: 10, gen2: 1
val it : bool = false
```

The problem with this version is that a huge amount of tasks are created and the administration of these tasks cancels out the advantage with multiple core.

This problem is handled by the introduction of a maximal depth to which new tasks are created:

```
let rec parExistsDepth p t n =
  if n=0 then exists p t
  else match t with
       | Leaf                  -> false
       | Node(_,v,_) when p v -> true
       | Node(tl,_,tr)         ->
           let b1 = Task.Factory.StartNew(
                          fun () -> parExistsDepth p tl (n-1))
           let b2 = Task.Factory.StartNew(
                          fun () -> parExistsDepth p tr (n-1))
           b1.Result||b2.Result;;
val parExistsDepth : ('a -> bool) -> BinTree<'a> -> int -> bool
```

Experiments show that the best result is obtained using depth 4:

```
parExistsDepth (fun n -> isPrime n && n>10000) t 4;;
Real: 00:00:35.303, CPU: 00:02:18.669,
GC gen0: 0, gen1: 0, gen2: 0
```

The speedup is approximately 2.3. At depths starting from about 22 the degradation of performance grows fast. This is not surprising taking the number of subtrees at such depths into account.

Example: Quick sort

A classical algorithm that is based on the divide and conquer problem-solving technique is the Quick sort algorithm that was developed by C.A.R. Hoare. The basic idea is very simple. The array:

$$
\begin{array}{r|c|c|c|c|c|}
 & \multicolumn{5}{c}{\overbrace{\hspace{12em}}^{\text{To be sorted}}} \\
\text{Indices :} & 0 & 1 & & n-2 & n-1 \\ \hline
\text{Values :} & v_0 & v_1 & \cdots\cdots & v_{n-2} & v_{n-1}
\end{array}
$$

is sorted by first rearranging the elements $v_1 \ldots v_{n-2}v_{n-1}$ such that the resulting elements $v'_1 \ldots v'_{n-2}v'_{n-1}$ can be partitioned into two sections with indices $1, \ldots, k$ and $k+1, \ldots, n-1$, respectively, such that all the elements in first section are smaller than v_0 and all the elements in the second section are greater than or equal to v_0:

$$\begin{array}{l|c|c|c|c|c|c|c|c|c}
\text{Indices :} & 0 & 1 & \cdots & k-1 & k & k+1 & \cdots & n-2 & n-1 \\
\hline
\text{Values :} & v_0 & v'_1 & & v'_{k-1} & v'_k & v'_{k+1} & & v'_{n-2} & v'_{n-1}
\end{array}$$
$$\underbrace{v'_1 \;\cdots\; v'_{k-1} \;\; v'_k}_{\text{All elements} < v_0} \quad \underbrace{v'_{k+1} \;\cdots\; v'_{n-2} \;\; v'_{n-1}}_{\text{All elements} \geq v_0}$$

The element v_0 can now be correctly placed in its final position by swapping it with the k's element:

$$\begin{array}{l|c|c|c|c|c|c|c|c|c}
\text{Indices :} & 0 & 1 & \cdots & k-1 & k & k+1 & \cdots & n-2 & n-1 \\
\hline
\text{Values :} & v'_k & v'_1 & & v'_{k-1} & v_0 & v'_{k+1} & & v'_{n-2} & v'_{n-1}
\end{array}$$
$$\overbrace{\underbrace{v'_k \;\; v'_1 \;\cdots\; v'_{k-1}}_{\text{To be sorted}}}^{\text{All elements} < v_0} \quad v_0 \quad \overbrace{\underbrace{v'_{k+1} \;\cdots\; v'_{n-2} \;\; v'_{n-1}}_{\text{To be sorted}}}^{\text{All elements} \geq v_0}$$

This array has the property that any element in the first section is smaller than any element in the second section, as the elements in the first section are $< v_0$ while the elements in the second section are $\geq v_0$. The array can hence be sorted by sorting each of the sections separately. This algorithm will have an average run time that is proportional to $n \cdot \log n$ and a worst-case run time proportional to n^2, where n is the length of the array.

The sorting algorithms available in the libraries have a better worst-case run time (proportional to $n \cdot \log n$) and they are using very efficient algorithms. So our recommendation is to use these libraries. We just use the Quick sort algorithm here to illustrate that the above method for parallelizing a divide and conquer algorithm applies to a non-trivial algorithm.

The function `swap` exchanges two elements of an array:

```
let swap (a: 'a[]) i j =
    let v = a.[i]
    a.[i] <- a.[j]
    a.[j] <- v;;
val swap : 'a [] -> int -> int -> unit
```

and the function `partition` can rearrange a section of an array:

$$\begin{array}{l|c|c|c|c|c|c}
\text{Indices :} & \cdots & k_1 & k_1+1 & \cdots\cdots & k_2 & \cdots \\
\hline
\text{Values :} & & v_{k_1} & v_{k_1+1} & & v_{k_2} &
\end{array}$$

so that the elements in the section which are smaller than a give value v comes before the elements which are greater than or equal to v:

$$\begin{array}{l|c|c|c|c|c|c|c|c|c|c}
\text{Indices :} & \cdots & k_1 & k_1+1 & \cdots & K & K+1 & \cdots & k_2 & \cdots \\
\hline
\text{Values :} & & v'_{k_1} & v'_{k_1+1} & & v'_K & v'_{K+1} & & v'_{k_2} &
\end{array}$$
$$\underbrace{v'_{k_1} \;\; v'_{k_1+1} \;\cdots\; v'_K}_{\text{All elements} < v} \quad \underbrace{v'_{K+1} \;\cdots\; v'_{k_2}}_{\text{All elements} \geq v}$$

The value of the expression `partition` $a\, v\, k_1\, k_2$ is K, that is, the index of the last element in the first section containing elements smaller than v:

```
let rec partition (a:'a[]) v k1 k2 =
    if k2=k1-1 then k2                         //empty section
    else if a.[k2] >= v then partition a  v k1 (k2-1)
         else swap a k1 k2
              partition a v (k1+1) k2;;
val partition : 'a [] -> 'a -> int -> int -> int
                  when 'a : comparison
```

The basic Quick sort algorithm is declared as follows:

```
let rec qsort a i j =
    if j-i>1 then let k = partition a a.[i] (i+1) (j-1)
                  swap a i k
                  qsort a i k
                  qsort a (k+1) j;;
val qsort : 'a [] -> int -> int -> unit when 'a : comparison

let sort a = qsort a 0 (Array.length a);;
val sort : 'a [] -> unit when 'a : comparison
```

So far we have just achieved an imperative program that can sort an array:

```
let a1 = [|1; -4; 0; 7; 2; 3|];;
val a1 : int [] = [|1; -4; 0; 7; 2; 3|]

sort a1;;
val it : unit = ()

a1;;
val it : int [] = [|-4; 0; 1; 2; 3; 7|]
```

Even though Quick sort is an imperative algorithm that changes an array, this does not cause any problems for a parallel version since the two recursive calls of `qsort` work on non-overlapping sections of the array – these two recursive call are independent of each other. Therefore, a parallel version that creates tasks up to a certain depth only is straightforwardly achieved using the same technique as used for the parallel search in a binary tree:

```
let rec pqsort a i j depth =
    if j-i<= 1 then ()
    else if depth=0 then qsort a i j
        else let k = partition a a.[i] (i+1) (j-1)
             swap a i k
             let s1 = Task.Factory.StartNew
                           (fun () -> pqsort a i k (depth-1))
             let s2 = Task.Factory.StartNew
                           (fun () -> pqsort a (k+1) j (depth-1))
             Task.WaitAll[|s1;s2|];;
val pqsort : 'a [] -> int -> int -> int -> unit
               when 'a : comparison
```

```
let parSort a d = pqsort a 0 (Array.length a) d;;
val parSort : 'a [] -> int -> unit when 'a : comparison
```

Since `pqsort` is an imperative algorithm we need to wait for the termination of both of the tasks `s1` and `s2` for the recursive calls. The function

```
Task.WaitAll: Task [] -> unit
```

is used for that purpose. It waits until all the provided tasks have completed their executions. Experiments show a speed-up of approximately 1.7 when sorting an array of size 3200000:

```
let a32   = Array.init 3200000 (fun _ -> gen 1000000000);;
let a32cp = Array.copy a32;;

sort a32;;
Real: 00:00:14.090, CPU: 00:00:14.024,
GC gen0: 1009, gen1: 3, gen2: 0
val it : unit = ()

parSort a32cp 7;;
Real: 00:00:08.352, CPU: 00:00:20.030,
GC gen0: 1016, gen1: 1, gen2: 1
val it : unit = ()
```

It is not surprising that `parSort` gets a smaller speed-up than `parExistsDepth`. The recursive call of `parExistsDepth` requires only that the disjunction `||` is computed on the values of the recursive calls, and this sequential part is a very fast constant-time operation. On the other hand, prior to the two recursive and parallel calls of `parSort`, a partitioning has to be made of the section to be sorted, and this sequential component has a run time that is linear in the size $(j - i)$ of the section.

Summary

In this chapter we have introduced

- asynchronous, reactive programs spending most of the wall-clock awaiting a request or a response from an external agent, and
- parallel programs exploiting the multi-core processor of the computer.

The common challenges and pitfalls in parallel programming are described. The `async` computation expression is introduced and it is shown how asynchronous computations can be used to make reactive, asynchronous programs with a very low resource demand. Library functions for parallel programming are introduced and it is show how they are used in achieving computations executing concurrently on several cores.

Exercises

13.1 Make program producing the deadlocked situation described on Page 315.

13.2 Make a type extension (cf. Section 7.4) of the class `AsyncEventQueue<'T>` with an extra member `Timer: 'T -> int -> unit` such that evaluating

```
evnq.Timer evnt n
```

will start an asynchronous computation that first sleeps `n` milliseconds and afterwards sends the event `evnt` to the queue `evnq`.

Hint: Apply `Async.StartWithContinuations` to `Async.Sleep` with suitable continuations.

13.3 Consider the dialogue program in Table C.1. Sometimes it is more convenient to let the functions for the state of the automaton communicate using shared variables rather than using function parameters. Revise the program so that `loading` and `finished` become parameterless functions. Is this revision an improvement?

13.4 Make a quiz program where a user should guess a number by asking the following questions:

- Is the number $< n$?
- Is the number $= n$?
- Is the number $> n$?

where n is a integer. The program can give the following answers:

- Yes
- No
- You guessed it!

The program must fix a random number between 0 and 59 to be guessed before starting the dialogue, and each run of the program should give a new number to be guessed.

13.5 Make a geography program guessing a country in Europe. The program asks questions to the user who answers `yes` or `no`. The program should use a binary tree with country names in the leaves and with a question in each node, such that the left subtree is chosen in case of answer `yes` and the right in case of answer `no`.

The program can be made to look more "intelligent" by inserting some random questions in between the systematic questions taken from the tree. The random questions should be of two kinds: Silly questions where the answer is not used by the program, and direct questions guessing a specific country where the answer is used by the program in case it gets answer `yes`.

13.6 The game of Nim is played as follows. Any number of matches are arranged in heaps, the number of heaps, and the number of matches in each heap, being arbitrary. There are two players A and B. The first player A takes any number of matches from a heap; he may take one only, or any number up to the whole of the heap, but he must touch one heap only. B then makes a move conditioned similarly, and the players continue to take alternately. The player who takes the last match wins the game.

The game has a precise mathematical theory: We define an operator xorb for non-negative integers by forming the *exclusive or* of each binary digit in the binary representation of the numbers, for example

$$\begin{array}{rcll} 109 & = & 1101101_2 & \\ 70 & = & 1000110_2 & \\ 109 \text{ xorb } 70 & = & 0101011_2 & = \quad 43 \end{array}$$

The xorb operator in F# is written `^^^`, for example:

```
109 ^^^ 70;;
```

```
val it : int = 43
```

The operator xorb is associative and commutative, and 0 is the unit element for the operator. Let the non-negative integers $a_1, \ldots, a_n$ be the number of matches in the n heaps, and let m denote the integer:

$$m = a_1 \text{ xorb } a_2 \text{ xorb } \cdots \text{ xorb } a_n$$

The following can then be proved:

1. If $m \neq 0$ then there exists an index k such that $a_k \text{ xorb } m < a_k$. Replacing the number a_k by $a_k \text{ xorb } m$ then gives a new set of a_i's with $m = 0$.
2. If $m = 0$ and if one of the numbers a_k is replaced by a smaller number, then the m-value for the new set of a_i's will be $\neq 0$.

This theory gives a strategy for playing Nim:

1. If $m \neq 0$ before a move, then make a move to obtain $m = 0$ after the move (cf. the above remark 1).
2. If $m = 0$ before a move, then remove one match from the biggest heap (hoping that the other player will make a mistake, cf. the above remark 2).

Use this strategy to make a program playing Nim with the user.

13.7 In this exercise we shall use data and task parallelism in connection with computation of Fibonacci numbers.

- Consider your solution to Exercise 1.5. Make an array containing the integers from 0 to 40 and apply `Array.map` to compute the first 41 Fibonacci numbers. Measure the run time of this operation. Make a data parallel solution using `Array.Parallel.map` and compare the solutions.
- Make a task-parallel solution to Exercise 1.5 and measure the obtained speedup when computing big Fibonacci numbers.
- Compare the obtained results with a sequential solution using accumulation parameters (Exercise 9.7). Note that the linear speedup obtained using multiple cores does not replace the use of good algorithms.

13.8 In this exercise you shall make a list-based version of the Quick sort algorithm. Note that you can use `List.partition` in your solution. Make a sequential as well as a task-parallel version and measure the speedup obtained by using the parallel version. The speedup for the list-based version should be smaller than that for the array-based version due to garbage collection (that is a sequential component) and due to the sequential operation that appends two sorted sub-lists.