# Monadic Scripting in F# for Computer Games

G. Maggiore, M. Bugliesi, R. Orsini

Università Ca' Foscari Venezia
{maggiore,bugliesi,orsini}@dais.unive.it

**Abstract.** Modern game architectures provide for a clean separation between game engine and game content, making engines *scriptable*, so as to delegate the development of the main functionalities related to gameplay to scripts coded in high-level languages.
Game scripting poses two orders of problems. First, the scripting language must be equipped with coroutining mechanisms to support a smooth interaction between the discrete-time structure of the game animation implemented by the engine with the execution of the scripts, which typically implement character behavior spanning multiple simulation ticks. Secondly, these mechanisms should be transparent, to ease design and development, and at the same time offer the highest run-time performance to keep up with the interactive frameratess expected by the user.
We present a monadic framework that elegantly solves these problems and compare it with the most commonly used systems, and discuss how it has been integrated in actual projects.

**Keywords:** games, monadic programming, state management, scripting

## 1 Introduction

Computer games promise to be the next frontier in entertainment, with game sales having topped movie and music sales in 2010 [4]. This unprecedented market prospects and potential for computer-game diffusion among end-users has created substantial interest in research on principled design techniques and on cost-effective development technologies for game architectures. Our present endeavour makes a step along these directions.

The architecture of a computer game is centered around the game's underlying discrete simulation engine [10], a complex piece of software which provides the core functionalities related to the *game content* – most notably the game logic, determining the behavior of the various game characters – and at the same time acts as the central coordination unit among the different components of the architecture – the rendering and graphic algorithms and functions, the physics engine, the input-output/networking devices, ... and so on.

Given their inherent complexity, game engines are usually written in low-level languages in order to ensure the high performance figures required for a smooth playing experience. Unfortunately, the use of such languages has two main drawbacks. First, game logic and AI require very frequent changes and fine tuning throughout all phases of game development. As such, coding gameplay functionalities within the engine implies long (re)compilation times and hence heavy design/development costs. Secondly, game logic and AI are built, modified and maintained by game designers, who are often rather inexperienced with computer programming concepts and techniques and hence may hardly operate effectively with low-level programming languages.

To counter these shortcomings, modern game architectures tend to support a clean separation between the development of the game engine and its content, with engines made *scriptable*, so that their gameplay functionalities can be programmed without direct access to the engine internals and with a higher-level language. This is typically achieved by embedding into the game engine the virtual machine of the chosen language and by exporting the engine itself as an API for that language. Scripting then provides the ability to code major game functionalities directly within code snippets (the scripts) that are interpreted by the game engine, without touching the engine at all. This makes games highly customizable and at the same time keeps the development costs under control by drastically reducing compilation time and greatly simplifying the scripting experience. Games scripting has become common practice, to the point that most commercial games nowadays come integrated either with off-the-shelf scripting languages, most notably LUA [6], Python [11] or C# [7], or else with in-house built languages [9].

In the present paper, we propose a novel, statically typed game scripting language based on a monadic DSL [20] built on top of F# [13]. Our DSL combines the flexibility of programming abstractions comparable to those offered by LUA with the benefits of strong, static typing. In addition, our DSL language supports a rather smooth integration between the execution model of the simulation engine, based on discrete-time updates of the game state, and the logic implemented by the scripts, which typically encode actions that span multiple update time-slots. As in other scripting languages, this integration is achieved by equipping our DSL with coroutines, which we encode within the monadic operators for binding and return. As a result our DSL offers greater flexibility over coroutines which are wired inside the virtual machine itself; this flexibility, in turn, makes it possible to tailor our scripting system precisely around the requirements of the game, without knowledge about the (complex) internals of a virtual machine. Also, encapsulating coroutines inside a monad effectively makes them mostly transparent to the developer. Interestingly, the additional flexibility comes with very limited overhead: indeed, our scripts run faster than LUA's and at least as fast as C# scripts. We have tested extensively our scripting system by using it in many projects, among which the most important is arguably [5], an upcoming strategy game that uses our scripts for managing most of its game logic, animating the world entities, handling user input and even implementing the networking subsystem. Our scripting system also lends itself to be programmed through a visual editor for non-technical users, such as gameplay designers and others.

*Paper structure* Section 2 gives a brief review of game architectures and of the technical issues arising in scriptable engines. Section 3 details the core runtime of our DSL, while Section 4 develops the DSL itself as a library of combinators built around the core that support a powerful set of customized behaviors, and illustrates an actual example of our DSL at work in the development of a game. Section 5 presents our results, Section 6 discusses the introduction of a visual environment to create scripts for non-technical users and Section 7 concludes the presentation.

## 2 Game Engine Architectures

In this section we briefly illustrate a (heavily) simplified game architecture in order to describe the main problems that a scripting system faces when introduced inside a game engine.

A game engine is based on three fundamental components: (*i*) the game state, which is a snapshot of the game world and includes a description of all the various entities of the game; (*ii*) the update function, which computes the next value of the game state, and (*iii*) the draw/rendering function, which draws the game state to the screen.

The game loop is the code that defines how the game is run; it is a recursive function that continuously invokes the update and draw functions with the current state as input. The game loop also computes the time delta between iterations, so that the update function will be able to adjust its computations to cover the actual amount of time elapsed between its invocations:

```
let run_game (game:Game) t =
  let t' = get_time()
  do game.Update (game.State, t'-t, t)
  do game.Draw (game.State)
  do run_game game t'
```

Central to our present concerns is the update function, which implements all the functionalities that modify the game state. As discussed earlier, most of these functionalities, typically the physics of the various entities, such as forces, collision detection, ... etc, the interaction with the input/output and other devices are coded in low-level languages such as C or C++ to guarantee the fast framerates needed for a smooth play experience. On the other hand, higher-level aspects of the game, related to gameplay, are typically left outside the code of the update function, and made scriptable.

The most important function of the scripts is to model the behaviors of the computer characters and of the other in-game objects. To illustrate, the following pseudo-code describes the behavior of a prince in a Role Playing Game:

```
prince:
  princess = find_nearest_princess()
  walk_to(princess)
  save(princess)
  take_to_castle(princess)
```

The main problem in coding this behavior with a script is to achieve a smooth interaction between the discrete-time structure of the game animation implemented by the simulation engine, and the behavior implemented by the script, which spans multiple time slots of the simulation engine. Specifically, in order to guarantee a smooth user experience, each such script must be interruptible, so that at each discrete step of the simulation engine the script performs a finite number of transitions and then suspend itself: failing to do so would slow down the simulation steps, hence the resulting framerate of the game would decrease, thereby reducing the player immersion.

The problem is sometimes addressed by coding scripts as state machines (SMs), whose execution gets interrupted at each state transition. However, while SMs represent a viable design choice for simple scripts, they are far less effective for modelling objects with complex behavior, as their structure grows easily out of control and becomes rather hard to maintain. Modern scripting languages adopt *coroutines* as a mechanism to build state machines implicitly, by way of their (the coroutines') built-in mechanisms to suspend and resume execution. With coroutines the code for a SM is written "linearly" one statement after another, but each action may suspend itself (an operation often called "yield") many times before completing. The local state of the state machine is stored in the continuation of the coroutine. Some of the most used scripting languages, which are Lua, Python and C#, all offer some suspension mechanisms similar to coroutines that game developers use for scripting; for a detailed discussion of couroutines in these languages, see [16, 12, 2].

## 3 The Script Monad

Monads can be used for many purposes [15, 18, 17, 19, 1, 14]. Indeed, monads allow us to overload the bind operator, in order to define exactly what happens when we bind an expression to a name. We will use this capability of monads to implement a DSL for coroutines that allows to chain coroutines together with the binding operator. The monad we define will *suspend* itself at every bind and return its continuation as a lambda. The monad type is `Script`:

```
type Script<'a,'s> = 's -> Step<'a,'s>
 and Step<'a,'s> = Done of 'a | Next of Script<'a,'s>
```

Notice that the signature is very similar to that of the regular state monad, but rather than returning a result of type $\alpha$ it returns either `Done of` $\alpha$ or the continuation `Next of Script<`$\alpha, \sigma$`>`. The continuation stores, in its closure, the local state of a suspended script. Our monad allows us to access the state which will be passed by the game engine; this way our scripts will be able to read, write or modify the main state of the game to interact with the processing performed by the game engine.

Returning a result in this monad is simple: we just wrap it in the `Done` constructor since obtaining this value requires no actual computation steps. Binding together two statements is more complex. We try executing the first statement; if the result is `Done x`, then we return `k x s`, that is we perform the binding and we continue with the rest of the program with the result of the first statement plugged in it. If the result is `Next p'`, then we cannot yet invoke `k`. This means that we have to bind `p'` to `k`, so that at the next execution step we will continue the execution of `p` from where it stopped.

```
type ScriptBuilder() =
  member this.Bind(p:Script<'a,'s>, k:'a->Script<'b,'s>)
    : Script<'b,'s> =
    fun s ->
      match p s with
      | Done x -> k x s
      | Next p' -> Next(this.Bind(p',k))

  member this.Return(x:'a) : Script<'a,'s> = fun s -> Done x

let script = ScriptBuilder()
```

We now define the `get_state` coroutine that extracts the current state:

```
let get_state : Script<'s,'s> = fun s -> Done s
```

We also define another coroutine that forces a suspension:

```
let suspend :Script<Unit,'s> = fun s -> Next(fun s -> Done())
```

We assume the standard F# convention that `let! x = script1 in script2` is translated into `Bind(script1, fun x -> script2)` and `return x` is translated into `Return(x)`.

Let us now see a small, self-contained example of our scripting system in action. Coroutines can be used in many ways to achieve various results; what we are more interested in, is using coroutines as a means to perform long and complex computations asynchronously inside the main loop of an application. We wish to build an application that computes a very large Fibonacci number, but does so while continuously writing on the console that it is still alive and responsive. This application has no shared state, and so all our coroutines will have type `Script<'a,Unit>`.

The coroutine version of the Fibonacci function is very similar to a regular implementation of the Fibonacci function, with the only difference that we use monadic binding to recursively invoke the function itself. Each time we recurse, the coroutine suspends:

```
let rec fibonacci n : Script<int,Unit> =
  script{
    match n with
    | 0 -> return 0
    | 1 -> return 1
    | n ->
      do! suspend
      let! n1 = fibonacci (n-1)
      let! n2 = fibonacci (n-2)
      return n1+n2
  }
```

Running the Fibonacci function now requires many steps of our scripting monad; for this reason we can safely invoke this function with the knowledge that it will run for a short time before returning either the final result with `Done` or its continuation with `Next`. We can define the main loop of our application as follows:

```
let main_loop() =
  let rec main_loop (f:Script<int,Unit>) =
    do printf "I␣am␣alive.\n"
    match f () with
    | Done result ->
      do printf "The␣result␣is␣%d\n" result
    | Next f' ->
      main_loop f'
  do main_loop (fibonacci 1000000)
```

The main loop above can be seen as a simplification of the game loop we have seen in Section 1. Of course in a game the state datatype would be more complex than just `Unit`, and the application would perform its full update and draw instead of just printing a string on the screen. This said, adding the above pattern matching to each iteration of the main loop of the game is all that is required to integrate our scripting system with an existing game engine.

## 4 A DSL for Scripting

The script monad is the runtime core of our DSL. A DSL can be augmented by defining a series of operators that automate or simplify common operations for the DSL developers. The best set of operators for a scripting DSL is strongly dependent upon the kind of game to be scripted. In this section we describe a general-purpose set of operators that make up a basic calculus of coroutines, but we would expect that other game developers would define additional operators that are a tighter fit to their games. The operators of our calculus of coroutines take as input one or more coroutines and return as output a new coroutine:

- `parallel` ($s_1 \wedge s_2$) executes two scripts in parallel and returns both results
- `concurrent` ($s_1 \vee s_2$) executes two scripts concurrently and returns the result of the first to terminate
- `guard` ($s_1 \Rightarrow s_2$) executes and returns the result of a script only when another script evaluates to `true`
- `repeat` ($\uparrow s$) keeps executing a script over and over
- `atomic` ($\downarrow s$) forces a script to run in a single tick of the *discrete simulation engine*

We show here the implementation of one of these combinators with our monadic system. To see the other implementations, see [13].

```
let rec parallel_ (s1:Script<'a>) (s2:Script<'b>)
          : Script<'a * 'b> =
  fun s ->
    match s1 s,s2 s with
    | Done x, Done y    -> Done (x,y)
    | Next k1, Next k2  -> parallel_ k1 k2
    | Next k1, Done y   -> parallel_ k1 (fun s -> Done y)
    | Done x, Next k2   -> parallel_ (fun s -> Done x) k2
```

We can now give another self-contained example that shows a producer and a consumer running in parallel. We start by defining the state as a single memory location, which can either be empty (`None`) or contain a value (`Some x`):

```
type Buffer = { mutable Contents : Option<int> }
let buffer = { Contents = None }
```

We define some additional helper functions to access the state. A good engineering rule is that the more complex is the state, the less coroutines directly use the `get_state` function; rather, when the state is complex, we define a series of additional accessor functions that help us manipulating the various aspects of the state:

```
let set_buffer v : Script<Unit,Buffer> =
  script{
    let! s = get_state
    s.Contents <- Some v }
let is_buffer_empty : Script<bool,Buffer> =
  script{
    let! s = get_state
    return s.Contents = None }
```

We define the producer (the consumer is symmetric) as follows; notice that each access to the state automatically suspends the coroutine, so we do not need to explicitly invoke `suspend`:

```
let rec wait_buffer_empty:Script<Unit,Buffer> =
  script{
    let! c = is_buffer_empty
    if c = false then
      do! suspend
      do! wait_buffer_empty }

let producer =
  let rec producer i : Script<Unit,Buffer> =
    script{
      do! wait_buffer_empty
      do! set_buffer i
      do! producer (i+1) }
```

The main loop is very similar to the main loop seen for the Fibonacci sample above; the only difference is that we invoke the producer and the consumer coroutines in parallel, by passing to the inner `main_loop` function the value `parallel_ producer consumer`.

### 4.1 Scripting in Games

We now discuss the introduction of our scripting system in games. In the following we outline how we have built most of the game logic of the upcoming RTS game Galaxy Wars (which is also released with a fully open source [5]). In the game we are considering the players compete to conquer a series of *star systems* by sending fleets to reinforce their systems or to conquer the opponent's.

**Game Patterns** Thanks to our general combinators we can define a small set of recurring game patterns; by instantiating these game patterns one can build the actual game scripts with great ease. These patterns may be adapted for the specific domain of a game, or altogether new patterns may be created that better fit one's game. The first game pattern we see is the most general, and for this reason it is called `game_pattern`. This pattern initializes the game in a single tick, then performs a game logic (while the game is not over) and finally it performs the ending operation before returning some result. The initialization is performed by the `init` script, which returns a result of a generic type `'a`; this result is the state of the script, and contains data that may be helpful for tracking additional information that is useful to our scripts but which is not stored in the game state. The logic of the various game entities, such as their AI, is then performed, repeatedly, by the `logic` script. While the logic script is run, the `game_over` script continuously checks to see if the game has been won or lost and thus must be terminated; when the termination condition is met, the `ending` script is invoked that may show some recap of the game that has just ended. The game pattern is implemented as follows:

```
let game_pattern
      (init:Script<'a>)
      (game_over:'a -> Script<'bool>)
      (logic:'a -> Script<Unit>)
      (ending:'a -> Script<'c>)
      : Script<'c> =
  script{
    let! x = init |> atomic_
    let! (Left y) =
       concurrent_
         (guard_ (game_over x) (ending x))
         (logic x |> repeat_)
    return y }
```

Notice that with the introduction of appropriate operators we may remove many of the parentheses of the above sample if they are seen as a hindrance to readability. The main portion of the code above may then be rewritten as:

```
(game_over x => ending x) .|| (logic x |> repeat_)
```

The game pattern above is very general, but not all scripts always need all of its parameters. We can build less general game patterns by reducing the number of parameters; for example, we may build a game pattern that has no initialization, logic or ending sequence; such a game pattern would implement the case of a game script whose sole responsibility is to check the termination conditions for a game (those that trigger the "game over" screen):

```
let wait_game_over (game_over:Script<bool>) : Script<Unit> =
  let null_script = script{ return () }
  game_pattern
    null_script
    (fun () -> game_over)
    (fun () -> null_script)
    (fun () -> null_script)
```

Writing a script with our system will consist of instantiating one game pattern with specialized scripts as its parameters; these scripts will alternate accesses to the specific state of the game with invocations of combinators from the calculus seen above. In the next session we will see an example of this.

**A sample script** The state of the game contains a series of star systems, fleets, players and various other information:

```
type GameState = {
    StarSystems : StarSystem List
    Fleets      : Fleet List
    ... }
```

We define a series of accessor functions that allow a coroutine to access the current state of the game; such an accessor function, for example, is the `get_fleets` that returns the list of active fleets:

```
let get_fleets =
  script{
    let! state = get_state
    return state.Fleets }
```

The basic mode of the game uses our scripting system to determine the winner of the game; as long as there is more than one player standing, the script waits. This script computes the union of the set of active fleet owners with the set of system owners:

```
let alive_players_set =
  script{
    let! fs = get_fleets
    let fleet_owners =
             fs |> Seq.map (fun f -> f.Owner)
```

```
            |> Set.ofSeq
    let! ss = get_systems
    let system_owners =
            ss |> Seq.map (fun s -> s.Owner)
            |> Set.ofSeq
    return fleet_owners + system_owners }

let game_over =
  script{
    let! alive_players = alive_players_set
    let num_alive_players = alive_players |> Seq.length
    return num_alive_players = 1 }
```

The main task of our script is to wait until the set of active players has exactly one element; when this happens, that player is returned as the winner:

```
let basic_game_mode = wait_game_over game_over
```

The two short snippets above are all there is to the main game mode.

Variations of the game are soccer (one system acts as the ball which can be moved around), capture the flag, siege and others. We omit a detailed discussion of the other variations for reasons of space; the important thing to realize is that all of these variations have been implemented with the same simplicity of the scripts above, by instancing one game pattern with appropriate scripts which are built with a mix of combinators interspersed with accesses to the game state.

**Input Management** Another large subsystem where we have used our scripting system is input management. Input is divided into a series of pairs of scripts; each pair of scripts is separated by a guard: the first script performs an event detection, while the second performs an event response. Each pair of scripts is repeated forever, in parallel with all other scripts.

As an example, consider the following script that decides whether to launch ships or not against a target:

```
let input world =
  repeat_(
  script{
    if mouse_clicked_left() then
      let mouse = mouse_position()
      // find closest planet under the cursor
      let clicked:Option<Planet> = ...
      return Some(clicked) } =>
    fun p -> script{ world.SourcePlanet := Some(p) }) .||
  repeat_(
  script{
    if mouse_clicked_right() &&
       world.SourcePlanet <> None then
      let mouse = mouse_position()
      // find closest planet under the cursor
      let clicked:Option<Planet> = ...
      match clicked with
      | Some planet ->
        return Some(planet,world.SourcePlanet.Value)
      | None -> return None } =>
    fun (source,target) ->
      script{ mk_fleet world source target })
```

A distinct advantage of this technique is that it allows us to cleanly separate the code that reads the actual user input from the code that performs something meaningful on the game world with this input. By parametrizing the code above with respect to the input detection scripts we could make it possible to support different controller types (game pad, touch panel, mouse, keyboard, etc.).

**Further uses** Since our scripting system proved effective in the various areas where we tried it, and given that its performance is fully satisfactory, we have decided to use it more pervasively all over the game. The result is that each entity (planets, ships, players, etc.) has a large chunk of its game logic moved from the update loop into appropriate scripts.

Our menu system is heavily based on our scripts (this proved especially useful when implementing the multi-player lobby, where the regular menu (buttons, input detection, etc.) had to be run interleaved with the scripts that synchronize the list of players across the network.

Finally, we have used scripts for the entire networking system, given that many operations such as connections and time-outs require timers and many parallel operations.

## 5 Benchmarks

We will focus our comparison mostly on LUA, since it is the most widely adopted scripting language, it fully supports coroutines and is considered among the state of the art. We will include some benchmarking data on Python and C# for completeness, but their poor support for coroutines makes them unsuitable for large scale use as scripting languages. For a more detailed discussion of the mechanisms of coroutines in Lua, Python and C# see [16, 12, 2].

LUA and F# offer roughly the same ease of programming, given that:

- scripts are approximately as long and as complex
- there are no explicit types, thanks to dynamic typing in LUA and type inference in F#

It is important to notice that, since F# is a statically type language, it offers a relevant feature that LUA does not have: **static type safety**. This means that more errors will be catched at compile time and correct reuse of modules is made easier.

To measure speed, we have run three benchmarks on a Core 2 Duo 1.86 GHz CPU with 4 GBs of RAM. The tests are two examples of scripts computing large Fibonacci numbers concurrently plus a synthetic game where each script animates a ship moving in a level and then dying. The tests have been made with Windows 7 Ultimate 64 bits. Lua is version 5.1, Python is version 3.2 and .Net is version 4.0.

**Table 1.** Samples length

| Language | Test 1 | Test 2 | Test 3 |
|---|---|---|---|
| F# | 21 | 21 | 35 |
| Python | 24 | 29 | 48 |
| LUA | 30 | 39 | 52 |
| C# | 51 | 58 | 59 |

**Table 2.** Samples speed

| Language | Test 1 | Test 2 | Test 3 |
|---|---|---|---|
| Cnv | 7.6 | 5.8 | 4.0 |
| Python | 1.1 | 1.1 | 0.9 |
| LUA | 1.5 | 1.4 | 0.8 |
| C# | 7.1 | 4.2 | 4.1 |

We have measured the total length of each script (Table 1) to give a (very partial) assessment of the expressiveness of each solution, plus the number of yields per second (Table 2) in order to assess the relative cost of the yielding architecture; more yields per second implies more scripts per second which in turn implies more scripted game entities and thus a more complex and compelling game-play.

It is quite clear that F# offers the best mix of performance and simplicity. Also, it must be noticed that Python and Lua suffer a noticeable performance hit when accessing the state, presumably due to lots of dynamic lookups; this problem can only become more accentuated in actual games, since they have large and complex states that scripts manipulate heavily.

An additional note must be given about architectural convenience. For games where the `discrete simulation engine` is written in C# (either because the entire game is written in C# or because the game is written in C++ while only the game logic is in C#) then using a language such as F# can give a further productivity and runtime performance boost because scripts would be able to share the game logic type definitions given in C#, thereby removing the need for binding tools such as SWIG or the DLR [8, 3] (or many others) that enable interfacing C++ or C# with Lua or Python.

## 6 A Visual Editor (future work)

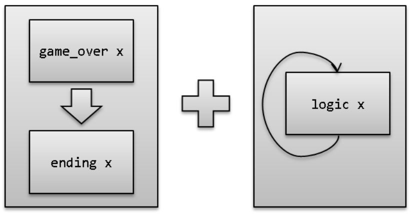
One of our main concerns is that our scripting system as presented so far is aimed at technical users. While this definitely offers advantages, since the game engine itself can be partially coded with scripts to the great benefit of the productivity of developers, the need for less technically inclined users such as designers to access the scripting system is very relevant. After all, designers play a very important role in shaping the behaviors and scripts of a game.

For this reason we are studying a visual editor that allows to create scripts without having to write source code. This editor (which is still a work in progress) would allow the user to drag blocks that represent operations and combinators of our scripts, and assemble them in a fashion that is similar to that of flow-charts.

One of the scripts that we have seen above:

```
(game_over x => ending x) .|| (logic x |> repeat_)
```

Would appear in our editor as:



## 7 Conclusions

Scripts are an important and pervasive aspect of computer games. Scripts simplify the interaction with computer game engines to the point that a designer or an end-user can easily customize gameplay. Scripting languages must support coroutines because these are a very recurring pattern when creating gameplay modules. Scripts should be fast at runtime because games need to run at interactive framerates. Finally, the scripting runtime should be as modular and as programmable as possible to facilitate its integration in an existing game engine.

In this paper we have shown how to use meta-programming facilities (in particular monads) in the functional language F# to enhance the existing scripting systems which are based on Lua, the current state of the art, in terms of speed, safety and extensibility. We have also shown how having a typed representation of coroutines promotes building powerful libraries of combinators that abstract many common patterns found in scripts. As evidence of the capabilities of our proposed system we have outlined a series of applications of our scripts into an actual game that is under development.

## References

1. C# async and await (reference). http://msdn.microsoft.com/en-us/vstudio/async.aspx.
2. C# yield (reference). http://msdn.microsoft.com/en-us/library/9k7k7cf0(v=vs.80).aspx.
3. Dlr (dynamic language runtime). http://dlr.codeplex.com/.
4. Entertainment software association. http://www.theesa.com.
5. Galaxy wars game. http://vsteam2010.codeplex.com/.
6. Games using lua as a scripting language. http://en.wikipedia.org/wiki/Category:Lua-scripted_video_games.
7. Scripting in unity. http://unity3d.com/support/ documentation/ScriptReference/index.Coroutines_26_Yield.html.
8. Swig (simplified wrapper and interface generator). http://www.swig.org/.
9. Unrealscript documentation. http://unreal.epicgames.com/UnrealScript.htm.
10. Lars Bishop, Dave Eberly, Turner Whitted, Mark Finch, and Michael Shantz. Designing a pc game engine. *IEEE Comput. Graph. Appl.*, 18:46–53, January 1998.
11. Bruce Dawson. Game scripting in python. http:// www.gamasutra.com/features/ 20020821/dawson_ pfv.htm, 2002. Game Developers Conference Proceedings.
12. Ana L. de Moura, Noemi Rodriguez, and Roberto Ierusalimschy. Coroutines in Lua. *Journal of Universal Computer Science*, 10(7):910–925, July 2004.
13. Giuseppe Maggiore and Giulia Costantini. *Friendly F# (fun with game programming).* Smashwords.
14. Erik Meijer, Brian Beckman, and Gavin Bierman. Linq: reconciling object, relations and xml in the .net framework. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, SIGMOD '06, pages 706–706, New York, NY, USA, 2006. ACM.
15. Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93:55–92, 1989.
16. Guido Van Rossum and Phillip Eby J. Pep 342 - coroutines via enhanced generators. http://www.python.org/dev/peps/pep-0342/, 2010.
17. Philip Wadler. Comprehending monads. In *Mathematical Structures in Computer Science*, pages 61–78, 1992.
18. Philip Wadler. How to declare an imperative. *ACM Comput. Surv.*, 29(3):240–263, 1997.
19. Philip Wadler and Peter Thiemann. The marriage of effects and monads, 1998.
20. Keith Wansbrough, Keith Wansbrough, John Hamer, and John Hamer. A modular monadic action semantics. In *In Conference on Domain-Specific Languages*, pages 157–170, 1997.