Monadic Scripting in F# for Computer Games

G. Maggiore, M. Bugliesi, R. Orsini

Università Ca’ Foscari Venezia
DAIS - Computer Science
{maggiore,bugliesi,orsini}@dais.unive.it

Abstract

Scripting in video games is a complex challenge as it needs to allow a game designer (usually not a developer) to interact with an extremely complex piece of software: a game engine. Game engines handle AI, physics, rendering, networking and various other functions; scripting languages usually act as the main interface to drive the entities managed by the game engine without exposing too much of the complexity of the engine.

A good scripting language must be very user-friendly because most of its users will not be developers; it must support transparent continuation mechanisms to ease the most common tasks faced when writing scripts and it must be easy to integrate in an existing game engine. Also, since games are very performance sensitive, the faster the scripting system the better.

In this paper we present a monadic framework that elegantly solves these problems and compare it with the most commonly used systems.

Keywords: games, monadic programming, state management, scripting

1 Introduction

Games are the next frontier in entertainment. Game sales in 2010 have reached 10 billion dollars, making games the absolutely preferred means of entertainment of our time. As more and more developers focus on building games, we believe that a contribution can be made to this field with a study of game engine architectures and languages for making games.

The core of a game is its engine. A game engine [12] is a fairly complex piece of software: it encompasses most of the aspects of computer game development, touching areas such as computer graphics, AI, algorithms, networking, and so on. Game engines are difficult to maintain and long to compile. Game designers need a simpler access venue to build the game logic of a game, and for this reason game engines are made scriptable, so that their functionality
related to gameplay can be programmed without direct access to its source and with a simpler language.

A general architecture of scriptable game engines (we report here Figure 2 from [23]) can be seen in Figure 1.

As we can see the discrete simulation engine joins together the processing of the game state performed by the AI engine, the physics engine, user input and various scripts. The discrete simulation engine is responsible for maintaining an updated state and it is usually written in a language such as C or C++. C# is also emerging as the language of choice for implementing portions of the game engine (see [7]) and for implementing entire independent games ([9] and [11] are widely adopted game frameworks based on C#).

The scripting solutions that games use nowadays are based on either simple, in-house built languages [10] or ready-made scripting languages: among those we find Lua and Python ([5,13]), together with C# which sits somewhere in between a proper scripting language and a game development language.

The main contribution of this work is an improvement over existing scripting languages, most notably LUA (the current state of the art). We use F# in combination with a monadic domain specific language [22] to create a statically typed scripting language that is as succint as LUA but which is, thanks to type safety, more robust. Also, encoding coroutines (one of the major characteristics of scripting languages for games) with monads offers greater flexibility over LUA’s approach of wiring coroutines inside the virtual machine itself; this flexibility 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. Finally, the runtime overhead of our system is so little that our scripts run faster than LUA’s and at least as fast as C# scripts.

Section 2 describes current approaches to scripting and explains the metrics we will use to compare our system with these approaches. Section 3 details the core runtime of our monadic scripting language, while Section 4 develops a library of useful combinators that are built around the core and support a powerful set of customized behaviors: this library of combinators represents...
our scripting language. Section 5 illustrates an actual example of how we have used our DSL in the development of a game. Section 6 presents our results and Section 7 concludes the presentation.

2 Scripting in Games

The most important function of scripts is that of modeling the behaviors of characters and other in-game objects; in the remainder of the paper we will focus on this specific aspect. As an example, consider a script that describes the behavior of a prince in an RPG game:

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

Depending on the size of the game world, there may be up to thousands of scripts running at any time. This means that each script must be interruptible, that is at each discrete step of the simulation engine each script must perform a finite number of transitions and then suspend itself; failing to do so would slow down the simulation steps, and the resulting framerate of the game would decrease, thereby reducing the player immersion. For this reason simple scripts are sometimes coded as a state machine (SM). As behaviors grow more complex, the code for an SM becomes difficult to maintain. Also, the designers who write these scripts think in terms of (nested) sequences of character actions and not in terms of SMs. For this reason scripting languages make heavy use of coroutines as a mechanism to build state machines implicitly instead of coding them explicitly as seen in the snippet above [6,10,14]. Coroutines are generalization of subroutines that allow multiple entry points for suspending and resuming execution at certain locations. With coroutines the code for a SM is written “linearly” one statement after another, but thanks to the suspension mechanism each action may suspend itself (often called “yield”) many times before completing. The resulting code will look more like the pseudo-code for prince, where the current state of the state machine is stored implicitly in the current continuation.

We can state the list of requirements that a good scripting system should have:

- support for coroutines
- ease of programming
- speed (games require very fast execution)
- extensibility of the coroutine framework (to better adapt it to the game engine)
2.1 Coroutines in action

In the remainder of this section we analyze coroutines in action in Lua. We will also briefly discuss how coroutines are emulated in Python and C# with generators. We will implement the pseudo-code taken from the prince sample seen above. For a more detailed discussion of the mechanisms of coroutines in Lua, Python and C# see [18,15,2].

Lua coroutines are based on the three functions coroutine.yield, coroutine.resume and coroutine.create which respectively pause execution of a coroutine, resume execution of a paused coroutine and create a coroutine from a function.

```lua
function walk_to(self,target)
    return coroutine.create(
        function()
            while(dist(self,target) > self.reach) do
                self.Velocity = towards(self, target)
                coroutine.yield()
            end
        end)
end

function prince(self)
    return coroutine.create(
        function()
            princess = bind_co(find_nearest_princess(self))
            bind_co(walk_to(self, princess))
            bind_co(save(self, princess))
            bind_co(take_to_castle(self, princess))
        end)
end
```

Notice that to invoke a coroutine we need to explicitly bind it with the bind_co function, which resumes a coroutine until it yields for the last time, when it returns the resulting value:

```lua
function bind_co(c)
    local s,r,r_old = true,nil,nil
    while(s) do
        r_old = r
        s,r = coroutine.resume(c)
        coroutine.yield()
    end
    return r_old
end
```

A similar mechanism to implement coroutines in Python makes use of generators.

A generator is a special routine that returns a sequence of values. However, instead of building an array containing all the values and returning them all at once, a generator yields the values one at a time; yielding effectively suspends the execution of the generator until the next element of the sequence is requested by the caller. Python generators may appear as a way to return
Maggiore, Bugliesi, Orsini

lazy sequences but they are powerful enough to implement coroutines. We can adopt the convention that a coroutine is actually a generator which yields a sequence of null (None) values until it is ready to return; the returned value will be yielded last.

```python
def walk_to(self, target):
    while(dist(self, target) > self.Reach):
        self.Velocity = towards(self, target)
        yield

...  
def prince(self):
    for princess in find_nearest_princess(self):
        yield
    for x in walk_to(self, princess):
        yield
    for x in save(self, princess):
        yield
    for x in take_to_castle(self, princess):
        yield
```

As in Python, C# supports generators. Since C# is statically typed, we need to assign a type to our coroutines. We have two alternatives; a coroutine that returns nothing (void) has type `IEnumerable`, that is it returns a sequence of `Objects` that are all null (a similar strategy is used by Unity, even though with unsafe casts [6]) and we can type a coroutine that returns a value of type `T` as `IEnumerable<T?>`, where `T?` is either `null` or an instance of `T`.

We omit the C# sample for brevity, and also because of its similarity with Python. Moreover, when compared with LUA generators to implement coroutines are quite cumbersome in a scripting language and indeed LUA is by far more used in games.

In the remainder of the paper we will present a different approach to coroutines, namely building a meta-programming abstraction (called monad) to implement coroutines in F#. We will discuss how our approach produces code which is faster and shorter than similar implementations in Lua, Python and C#. We will also discuss how our approach is very customizable, thanks to the fact that coroutines are not wired into the language runtime but rather we have defined them with our monad. Also, thanks to type inference the resulting scripts require no typing annotations. Finally (see Section 6 for the details), our system offers a good runtime performance and is type safe; this makes it suitable for large and complex scripts.

3 The Script Monad

Monads can be used for many purposes [17,20,19,21,1,16]. Indeed, monads allow us to overload the bind operator, in order to define exactly what happens when we bind an expression to a name.
Monads in F# enjoy syntactic sugar that simplifies their use. Monadic operators are inserted with the specialized keywords \textbf{let!} for bind and \textbf{return} for return.

For our present purposes, one extremely relevant use we can make of monads is as the basis of coroutines for our scripting system.

The script monad is not the actual scripting language. Rather, it is the runtime framework that we use to transparently support coroutines. In Section 4 we will see how we can define a library of functions which can be seen as additional keywords and operators for the resulting scripting language.

The monad we define, at every bind will \textit{suspend} itself and return its continuation as a lambda. This is one possible, very simple implementation of coroutines which does not feature an explicit \textbf{yield} operator. The monad type is \texttt{Script}:

\begin{verbatim}
|type Script<\textquotesingle a,\textquotesingle s> = \textquotesingle s -> Step<\textquotesingle a,\textquotesingle s>
|    and Step<\textquotesingle a,\textquotesingle s> = Done of \textquotesingle a
|    | Next of Script<\textquotesingle a,\textquotesingle s>
\end{verbatim}

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 \texttt{Done of $\alpha$} or the continuation \texttt{Next of Script<$\alpha$,\sigma>}. The continuation stores, in its closure, the current state of a suspended script.

Returning a result in this monad is simple: we just wrap it in the \texttt{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 \texttt{Done x}, then we return \texttt{Next(k x)}, that is we perform the binding and we will continue with the rest of the program with the result of the first statement plugged in it. If the result is \texttt{Next p'}, then we cannot yet invoke \texttt{k}. This means that we have to bind \texttt{p'} to \texttt{k}, so that at the next execution step we will continue the execution of \texttt{p} from where it stopped.

\begin{verbatim}
|type ScriptBuilder() =
|    member this.Bind(p:Script<\textquotesingle a,\textquotesingle s>,
|           k:'a->Script<\textquotesingle b,\textquotesingle s>)
|        : Script<\textquotesingle b,\textquotesingle s> =
|        fun s ->
|            match p s with
|            | Done x -> Next(k x)
|            | Next p' -> Next(this.Bind(p',k))
|    member this.Return(x:'a) : Script<\textquotesingle a,\textquotesingle s> =
|        fun s -> Done x
|let script = ScriptBuilder()
\end{verbatim}

Integrating our monadic runtime for scripts in a game engine loop is simple. We define a game script as an instance of the \texttt{Script} datatype where the state (the $\sigma$ type variable) is instantiated to some type \texttt{GameState} which defines the current state of the game. The main loop will now carry around the current
computation of the game script:

```ocaml
let rec update (script_step:Script<Unit,GameState>) (game_state:GameState) =
  let script_step' =
    match script_step game_state with
    | Done() -> fun _ -> Done()
    | Next k -> k
  let game_state' =
    (** compute new state **)
  in update script_step' game_state'
```

The update function executes a step of the script. If the script has finished, then we create an identity script that will be called indefinitely or we could return that the game is finished and some recap screen must be shown. When an iteration of the update loop is completed, then we call update with the next state of the script as its parameter.

This integration with the main loop can be easily translated to C# and then integrated with the rest of the game engine.

### Auxiliary Functions

Existing functions are, of course, not defined in terms of our monad. Often though, we will wish to apply some existing function directly to our scripts rather than bind them to some variables, apply the function to those variables and finally returning the result. For example, consider the case where we have two scripts `s1:Script<bool>` and `s2:Script<bool>` and we wish to compute the logical `and` of their result; currently we would have to write:

```ocaml
script{
  let! x = s1
  let! y = s2
  return x && y
}
```

whereas we would prefer to be able to simply write:

```
s1 &&. s2
```

for some appropriate operator `(&&.)`. For this reason we define the lifting functions, very useful functions that lift an operation from the domain of values to the domain of monads; the general shape of the n-ary lifting functions is:

```ocaml
let lift_n (f : 'a1 -> a2 -> ... -> 'an -> 'b) :
  (Script<'a1> -> Script<'a2> -> ... -> Script<'an> ->
   Script<'b>) =
  fun s1 -> s2 -> ... -> sn ->
  script{
    let! x1 = s1
    let! x2 = s2
    ...
    let! xn = sn
    return f x1 x2 ... xn
  }
```
Unfortunately it is very difficult to define $\text{lift}_n$ for an arbitrary $n$, so we will define $\text{lift}_i$ for various values of $i$. As an example application we can define binary operators for scripts:

```plaintext
let not_ (s:Script<bool>) : Script<bool> =
  lift_1 not s

let and_ (s1:Script<bool>) (s2:Script<bool>) : Script<bool> =
  lift_2 (&&) s1 s2

let or_ (s1:Script<bool>) (s2:Script<bool>) : Script<bool> =
  lift_2 (||) s1 s2
```

Also, we can define a useful $\text{ignore}_n$ function that discards the result of a script when we do not need it:

```plaintext
let ignore_ (s:Script<'a>) : Script<Unit> =
  lift1_ (fun x -> ()) s
```

In general any $n$-ary function that is not capable of manipulating scripts can be lifted to the domain of scripts with $\text{lift}_n$

## 4 A Library of Reusable Scripts

Here, and throughout we use the standard F# convention that, inside a monad, missing $\text{else}$ branches correspond to $\text{else}$ branches with $\text{return}()$ as the body. Similarly, we use the infix application operator $\mid\rangle$, writing $x \mid\rangle f$ as an equivalent for $(f x)$.

The main advantage of using monads rather than hardcoded mechanisms is flexibility. On one hand we can modify the definition of our monad in order to accommodate for different functionalities, such as referential transparency, multi-threading, etc. On the other hand we have an explicit representation of coroutines (values of type $\text{Script}$) with which we can easily build libraries that functionally manipulate coroutines in powerful ways. In this section we study one such general purpose library based on the script monad. This library is being used in the development of a commercial strategy game, and as such its usefulness has been put to the test in a practical application (the game is released as open source, and can be found at [4]). It is important to realize that even though what follows is a very general library of combinators (in particular the Calculus of Coroutines presented below) there are many alternative libraries that may better suit a specific kind of games; the monadic system described in Section 3 can be used as the basis for any of those alternative libraries.

A Calculus of Coroutines

The basic combinators we define are a simple calculus of coroutines; this means that with these operators we take one or more coroutines and we return
<table>
<thead>
<tr>
<th>name</th>
<th>syntax</th>
<th>operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>parallel</td>
<td>$s_1 \land s_2$</td>
<td>executes two scripts in parallel and returns both results</td>
</tr>
<tr>
<td>concurrent</td>
<td>$s_1 \lor s_2$</td>
<td>executes two scripts concurrently and returns the result of the first to terminate</td>
</tr>
<tr>
<td>guard</td>
<td>$s_1 \Rightarrow s_2$</td>
<td>executes and returns the result of a script only when another script evaluates to true</td>
</tr>
<tr>
<td>repeat</td>
<td>$\uparrow s$</td>
<td>keeps executing a script over and over</td>
</tr>
<tr>
<td>atomic</td>
<td>$\downarrow s$</td>
<td>forces a script to run in a single tick of the discrete simulation engine</td>
</tr>
</tbody>
</table>

Table 1

Calculus of Coroutines

another coroutine which can be plugged as a parameter for another one of this operators. The basic building blocks of these operators are instances of the script monad and are listed in Table 1.

We show here the implementation of these combinators with our monadic system:

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

let rec concurrent (s1:Script<'a>) (s2:Script<'b>) =
  fun s ->
  match s1 s, s2 s with
  | Return x, _ -> Return (Left x)
  | _, Return y -> Return (Right y)
  | Continue k1, Continue k2 -> concurrent_ k1 k2

let rec guard_ (c:Script<bool>) (s:Script<'a>) =
  script{
    let! x = c
    if x then
      let! res = s
      return s
    else
      let! res = guard_ c s
      return res
  }

let rec repeat_ (s:Script<Unit>) : Script<Unit> =
  script{
    do! s
    do! repeat_ s
  }

let rec atomic_ (p:Script<'a>) : Script<'a> =
```

43
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 final game scripts with great ease. The first game pattern 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:

```ocaml
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_ (victory x) (ending x |> atomic)) (logic x |> repeat_)
    return y
  }
```

A simplified, recurring variation of this game pattern simply does nothing until the game is over:

```ocaml
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.

5 An Actual Script

We are now ready to discuss an actual script coming from a strategy game. In this game, the players compete to conquer a series of systems by sending fleets to reinforce their systems or to conquer the opponent’s.

The basic game mode returns the winning player; 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:

```ocaml
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

An interesting alternative coding style could be built by lifting a series of useful operators and redefining `alive_players_set` and `game_over` in a very concise form; let us assume that all operators and functions that end with `_` or `.` in the following snippet have been lifted appropriately, for example:

let (=.) = lift_2 (=)
let (+.) = lift_2 (+)
...

This way, we can write:

let alive_players_set = (get_fleets |> Seq.map_ (fun f -> f.Owner) |> Set.ofSeq_) (+.)
let game_over = (alive_players_set |> Seq.length_) =. (script{ return 1 })

5.1 Other scripts

Variations of the game are soccer (one system acts as the ball which can be moved around), capture the flag, siege and others.

The siege game mode features a central system which must be conquered and held to obtain bonuses. Holding this system for sixty seconds gives a bonus to its owner, while losing the central system resets its bonuses.

The siege mode is an instance of the most general game pattern:

let wait dt = script{
  let! t0 = time
do! guard
    script{
      let! t = time
      return t - t0 > dt
    }
}
let init : Script<System> = get_system "Center"
let logic (center:System) =
  script{
    let previous_owner = center.Owner
    do! concurrent_
      script{
        do! wait 60.0f
        center.AddBonus()
      } (guard (script{ return center.Owner <> previous_owner })
      (script{
        center.ResetBonus()
        return ()
      }))
    |> ignore_
  }

let siege_game_mode =
  game_pattern init (fun center -> game_over)
  logic (fun _ -> script{ return () })

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.

6 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 the current 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.

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: safety. This means that more errors will be caught 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 syntetic 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. The lines of code of each script are listed in Table 2, while the number of yields per seconds (higher is better) are listed in Table 3. We have measured the number of yields per second 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 gameplay.

<table>
<thead>
<tr>
<th>Language</th>
<th>Fibonacci</th>
<th>Many Fibonacci</th>
<th>Ships</th>
</tr>
</thead>
<tbody>
<tr>
<td>F#</td>
<td>21</td>
<td>21</td>
<td>35</td>
</tr>
<tr>
<td>Python</td>
<td>24</td>
<td>29</td>
<td>48</td>
</tr>
<tr>
<td>Lua</td>
<td>30</td>
<td>39</td>
<td>52</td>
</tr>
<tr>
<td>C#</td>
<td>51</td>
<td>58</td>
<td>59</td>
</tr>
</tbody>
</table>

Table 2
Lines of code

<table>
<thead>
<tr>
<th>Language</th>
<th>Fibonacci</th>
<th>Many Fibonacci</th>
<th>Ships</th>
</tr>
</thead>
<tbody>
<tr>
<td>F#</td>
<td>7.6</td>
<td>5.8</td>
<td>4.0</td>
</tr>
<tr>
<td>C#</td>
<td>7.1</td>
<td>4.2</td>
<td>4.1</td>
</tr>
<tr>
<td>Lua</td>
<td>1.5</td>
<td>1.4</td>
<td>0.8</td>
</tr>
<tr>
<td>Python</td>
<td>1.1</td>
<td>1.1</td>
<td>0.9</td>
</tr>
</tbody>
</table>

Table 3
Speed test in millions of yields per seconds

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. See Figure 2 for a representation.
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 in terms of speed, safety and extensibility the existing scripting systems which are based on Lua, the current state of the art. We have also shown how having a typed representation of coroutines promotes building powerful libraries of combinator s that abstract many common patterns found in scripts.

References


