Università
Ca'Foscari
Venezia

Ca' Foscari
Dorsoduro 3246
30123 Venezia

Corso di Laurea magistrale
in Informatica

Tesi di Laurea

Design of an optimized compiler for Casanova language

Relatore
Chia.mo Prof. Agostino Cortesi
Correlatore
Dr. Giuseppe Maggiore

Laureando
Francesco Di Giacomo
Matricola 831569

Anno Accademico
2013/ 2014

Abstract

Making games is a complex, time-consuming and expensive task, since a game is made of several inter-operating components. For this reason developers having few resources at their disposal (i.e. researchers or independent developers) are discouraged in developing such games using standard languages (such as C++) and libraries (DirectX, OpenGL or XNA in .NET environment). The aim of this work is defining a language oriented to developing games which allows the aforementioned category of developers to realize a game or a real-time simulation in a short amount of time, with a more synthetic code and in a more expressive way, with a good trade-off on performances. We will start examining the older version of Casanova, which is interpreted, and we will examine its flaws and present an improvement. Then we will present the structure of the compiler itself, based on F# compiler. Our work shows the execution time improvement with respect to the older Casanova version, as well as the expressiveness and conciseness of the code with respect to an alternative implementation in the older version of Casanova 1.0.

Contents

Chapter 1

Introduction

Video game industry is a ever growing sector, whose profits have recently surpassed those of movies and music industry [5][1]. From this statement we could infer that the video game field is that of entertainment. However, video games are used as simulations in several *serious* environments, such as defence, education, scientific exploration, health care, emergency management, city planning, engineering, and politics [12]. Indeed, there exists a large documentation of the usage of games for serious purposes since the beginning of the 20th century, for example see [16].

Developing a game requires a large budget, which often is not available for small developers such as those of *independent games*, which are games sold generally at a lower price, and serious games [17]. The need of a large amount of funding comes from the fact that games are made of components called *game engines*. These engines are large and powerful tools which are hard to make and maintain over time[7]. This has brought game developers to implement their own game engine, and then re-use it for their next productions. This causes several issues, among which games developed by the same company tend to be very similar to each other, and successive versions of the same game at most extend the existing engine to implement a small number of new features[2].

For example, the game engine built for Neverwinter Nights, features a script language based on an extension of C, which is compiled by the game editor. This script system, however, does not allow the parallel execution of multiple scripts except in pre-defined situations. These scripts are called in parallel by events defined in the level file of the game, which are limited in numbers, such as a periodic execution every 1 second, when a player enters or exits an area,etc. The same engine has been used, with few modifications, in the next title Newerwinter Nights 2 [3].

Casanova is a language whose main aim is to ease the programmer from implementing a game engine from scratch, by providing primitives in the language such as time and synchronization statements, and an interface to a drawing engine. Indeed implementing such engines often requires to implement specific components, such as state machines described in Section 2.2, or writing boiler plate code to perform specific actions in the game. For example Casanova provides a generalized pattern for defining Real-Time Strategy Games[4], a script language defined using monads and a way to express continuous events as an approximation of differential equations[14].

In this thesis work we will face the problem of improving the current version of Casanova, which is interpreted, by building a compiler to achieve better performances. We will also design the new version of the language to extend its syntax with constructs typical of Object-Oriented languages, and we will try to simplify the internal scripting language, maintaining at the same time the same expressiveness.

We started by collecting feedback on programmers who used Casanova 1.0 to create video games samples (such as reproductions of *tetris* and *pac-man*). We identified Casanova 1.0 main flaws in having too many overlapping concepts, such as *rules* and *scripts*[12] (see Sections 2.1 and 2.2). We noticed that several users neglected the use of one of these components in favour of the other "abusing" of their function. We realized that we could merge these overlapping components into a single component having both functionalities. Based on this idea we redesigned Casanova 2.0 introducing *interruptible rules*, adding also some Object-Oriented structures equivalent to those available in the most widespread languages, such as Java, C++ and C#. We then identified that the main performance drawback was caused by the script language, implemented in Casanova 1.0 with monads, and by the fact that the language was interpreted. We decided to build a compiler based on the open-source version of the F# compiler for the following reasons:

- We can compile Casanova code in .NET byte code which is compatible with several game development libraries, such as XNA, Unity 3D or Monogame.

- Taking advantage of the similarity between Casanova and F# syntax definition, we can modify the lexer and parser to generate Casanova AST, and then simply translate it into F# AST. In this way we will not have to implement a type checker and a code generator because the back-end of F# will accept the translated AST.

- We can support interoperation between third-party .NET libraries and Casanova (see 4.1 for an example of a game using a .NET library to manage file input/output).

Our work shows that Casanova 2.0 achieves better performances than Casanova 1.0 in presence of a high number of game entities and manages a lot of time-consuming operations, such as detecting collisions among physical bodies, maintaining at the same time a high frame rate, while Casanova 1.0, even with optimizations through indices on queries not yet implemented in Casanova 2.0 fails to manage even a moderate amount of such events. Besides Casanova 2.0 has the same expressive power as Casanova 1.0, while the code complexity is reduced having merged scripts and rules in a common structure with the same properties and functionalities.

In Chapter 2 we will analyse in detail the first version of Casanova, introducing its main architecture and internal structure. Then we will expose its flaws and try to propose a new design eliminating them. We will present an informal overview of Casanova 2.0 and then give its formal syntax and semantics definition.

In Chapter 3 we will examine the compiler architecture, which is based on F# open-source compiler, describing the lexer/parser component, and finally an implementation of a state machine to manage the internal script engine.

In Chapter 4 we will show a fully-fledged game in Casanova 2.0, and then show a comparison between the benchmark sample written in Casanova 1.0 and Casanova 2.0.

Chapter 2

Casanova: a language for video games

Casanova 1.0 is a language thought to relieve programmers of dealing with the time and synchronization issues described below and to manage the drawing phase of the game. It uses the *RSD* (Rules-Scripts-Drawing) pattern: first we update the continuous component of the game through *rules*, then we manage the discrete component through *scripts* or *coroutines* and then we draw the scene.

In this chapter we will introduce the main components used in modern computer games and simulations. We start by giving a general overview on how these components are implemented in the language as *rules* and *coroutines*, and how they are used in a Casanova 1.0 program, by summarizing the more detailed work of G. Maggiore's PHD thesis [12] and article on monadic scripts [14]. We will then analyse their flaws and introduce the ideas behind Casanova 2.0, its main differences from its first version, the reasons behind the new language structure and finally the grammar definition and semantic rules. Since Casanova 1.0 rule and coroutine implementations use constructs typical of functional programming languages, we will briefly introduce them below in order to allow the reader to fully understand the technical details.

2.1 Structure of a computer game

A computer game is a real-time application which simulates a virtual environment. The user can interact with this environment and modify it through a set of actions. The game is usually made of two main components: the *logic engine*, which updates the entities and rules of the game, and the *graphics engine* which draws them after the previous update. These two components are called by a structure called *game loop*. Each iteration of the game loop is called *frame*.

The game loop keeps invoking the *update* function, which updates the game logic, and the *draw* function, which draws the game components. The update phase is made to reflect changes due to game logical rules (such as physics or user-uncontrollable entity interactions) or to user input interactions. The draw phase re-draws the game entities to reflect the changes made at the previous step (update). We know from multimedia video rendering techniques [10] that several times the next frame can be computed differentially with respect to the previous frame, in order to save computational time (and in this case also memory space). This method is based on the assumption that, in a video, there are only small changes between the current frame and the previous one if the sampling is high enough (excluding the case where objects pop up instantly in the next frame, but they rarely occurs). In computer games this assumption does not yield since we have several entities moving across the screen or special effects created to respond a user input (for instance in games with destructible environments, when a wall is destroyed, the fragments of the wall are actually entities "spawned" by the game engine and made fly around giving them a physical impulse), so keep tracking of which entities must be re-drawn is only a waste of computational time. For this reason it is used a method called *immediate mode*[9], where all the scene is re-drawn at every frame.

From the previous considerations, we can define the game structure in a F# -like syntax as follows[1]:

```
type Game =
  {
    Update : float -> Unit
    Draw : Unit -> Unit
  }
```

The game loop will be a function which takes as input a value of type `Game` and keeps invoking the `Update` and `Draw` functions at each iteration.

```
let game_loop (game : Game) : Unit =
  let rec main_loop (t : float) : Unit =
    let t' = getTime()
    let dt = t' - t
    do game.Update dt
    do game.Draw()
```

[1]Unit is a value used in functional programming which has a semantic similar to that of the `void` keyword used in imperative languages such as C or Java.

```
    do main_loop t'
  do main_loop (getTime())
```

From this definition we can infer that a change on a property of a game entity (such as its velocity, acceleration) could be expressed as a differential equation whose solution is numerically computed by subdividing the domain into intervals of size $h = \Delta t$ and using one of the several numerical algorithms to solve Ordinary Differential Equations (ODE).

For instance, let us consider a car accelerating at $K\ m/s^2$, with velocity function $v(t)$ and position function $p(t)$. The game loop updates the position and velocity at time $t = 0, t = h, t = 2h, ...$ The dynamics are described by the following system of differential equations:

$$\begin{cases} \dfrac{dp(t)}{dt} = v(t) \\ \\ \dfrac{dv(t)}{dt} = K \end{cases} \tag{2.1}$$

We can solve the previous system using the forward Euler method[15]. This method approximates the first derivative with the forward difference scheme, that is

$$\frac{df(t)}{dt} = \delta^{+}(t) = \frac{f(t+h) - f(t)}{h} \tag{2.2}$$

Substituting (2.2) in (2.1) we obtain (reminding that we set $h = \Delta t$):

$$\begin{cases} \dfrac{p(t+\Delta t) - p(t)}{\Delta t} = v(t) \\ \\ \dfrac{v(t+\Delta t) - v(t)}{\Delta t} = K \end{cases}$$

$$\begin{cases} p(t+\Delta t) = p(t) + v(t) \cdot \Delta t \\ \\ v(t+\Delta t) = v(t) + K \cdot \Delta t \end{cases}$$

From the previous example we can infer that the update phase of the game loop consists of solving the following Cauchy problem:

$$\begin{cases} \dfrac{dw(t)}{dt} = f(t, w(t)) \\ \\ w(t_0) = w_0 \end{cases} \tag{2.3}$$

where $w(t)$ is a function representing the state of the game, or *world*, at time t, and w_0 its initial state (i.e. the state generated when the data structure representing the world is created). From (2.3) we can write a pseudo-code for the car example updating its physical properties. The `World` contains the car position, velocity and acceleration (indeed our game is made of just one entity, which is our car). The update function will exploit the previous approximation updating the fields of our entity as follows:

```
type World =
  {
    CarPos : Vector2
    CarVel : Vector2
    CarAcc : Vector2
  }

let update (world : World) (dt : float) =
  world.CarPos = world.CarPos + world.CarVel * dt
  world.CarVel = world.CarVel + world.CarAcc * dt
```

Euler method is very simple and computationally not expensive but it is not very accurate. In some situations it might be required that more accurate methods are employed, such as Ralston's method which is a 3-staged RK-method, whose formula is the following:

$$\begin{cases} K_1 = y(t) \\ K_2 = y(t) + \frac{1}{2}\Delta t K_1 \\ K_3 = y(t) + \frac{3}{4}\Delta t K_2 \\ y(t+\Delta t) = y(t) + \frac{2}{9}\Delta t K_1 + \frac{1}{3}\Delta t K_2 + \frac{4}{9}\Delta t K_3 \end{cases} \tag{2.4}$$

Method (2.4) is more accurate than Euler's but it is computationally more expensive, since at each frame it must compute $K_i,\ i = 1, 2, 3$ and then update the function $y(t)$. This may cause the game loop to process less frames per second, resulting in a less smooth game rendering and causing a phenomenon called *stuttering*, where the user experiences an abrupt update of the frame. Usually a trade-off between precision and accuracy must be performed depending on the situation: if the simulation does not require a high numerical precision but a smooth rendering then we opt for simpler but faster numerical methods, while if the simulation requires a high numerical precision we choose more accurate numerical methods but the processed frames per second might be much lower. Usually I/O-bound simulations, such as Real-Time Strategy games or other kinds of games where the response to user inputs must be quick, favour higher frame rates instead of accuracy, while non-interactive simulations (such as a simulator which simulates the flow of a liquid inside a pipe) where just small adjustments are made favours precision instead of frame rate. More generally one should choose a less computationally expensive method as long as the visual artefacts, due to a rough approximation of differential equations, are not perceivable by the user.

2.2 Synchronization and state machines

The previous section showed how a game deals with continuous events, such as physical laws related to the behaviour of one or more entities. Games are also made of discrete events, which are triggered when a specific condition is verified or a certain amount of time has passed.

Let us consider a simple example where a user can activate a light in the world: the light is turned on if the user press `key1` and `key2` in sequence within 0.3 seconds. The first naive attempt to code this into the `Update` function could be implemented by a *busy waiting* technique, where the function indefinitely loops until the user presses `key1`, register the time when exiting the loop, then starts looping again waiting for `key2` to be pressed, measuring the time again after this condition is satisfied, and finally checking if the time difference is less than 0.3 seconds.

```
while (is_key_up(key1)) do sleep(0)
let t = getTime()
while (is_key_down(key2)) do sleep(0)
let t' = getTime()
let dt = t' - t
if dt < 0.3 then
  do light_on()
```

This solution has a major flaw due to the fact that the game loop is locked inside a while loop until a user performs the combination of those two keys. This will cause the rendering to stop because the update function will not terminate until the input allows it to break out of the loops. An attempt to improve this solution might be using a thread for each waiting conditions. Unfortunately, in a general case where the waiting conditions are not only 2 but arbitrary, the thread generation and management would waste too much memory and CPU time.

The traditional solution to this problem is to build a *state machine*, formalized by the following

Definition 2.1. A State Machine is a tern $M = (S, F, G)$ where

- $S = \{s_1, s_2, ..., s_n\}$ is a finite set of states.
- $F = \{f_{i,j} : i, j = 1, ..., n\}$ is a set of *transition functions* from s_i to s_j whose output determines the next state of the machine.
- $G = \{g_i : i = 1, ..., n\}$ a set of functions executed at the state s_i.

Informally, a state machine is made of a finite number of states, each pair of states is associated to a function which determines when the machine changes its state from s_i to s_j, and at each state transition, a function related to the current state is executed.

Going back to the previous example, our state machine will be made of 4 states: the first state is the one related to the first waiting condition (`key1` is pressed), the second one is related to the second waiting condition (`key2` is pressed within 0.3 seconds) and it is parametric with respect to time. The third and fourth are respectively `Success` or `Fail`, which represent the fact that the user manages successfully to hit the key combination within the given time. A F# pseudo-code implementation is given below:

```
type StateMachine =
  | Wait1
  | Wait2 of float
  | Success
  | Failure
```

The state machine is updated in the following way:

```
let transition (sm : StateMachine) (dt : float) =
  match sm with
  | Wait1 ->
    if is_key_down(key1) then Wait2(0.3)
    else Wait1
  | Wait2(t) ->
    if t <= 0.0 then Fail
    else if is_key_down(key2) then Success
    else Wait2(t - dt)
  | Success -> Success
  | Fail -> Fail
```

```
let state_function (sm : StateMachine) =
  match sm with
  | Success -> do light_on()
  | _ -> ()
```

As we can see from the code above, coding a simple statement such as ”To activate the light press `key1` then `key2` within 0.3 seconds” is actually a very hard task. Besides, the logic of the program itself is hidden behind a very complex structure such as a state machine. Furthermore, the pseudo-code above makes use of syntactical structures, such as *Discriminate Unions*, which are available only in functional languages. Writing the definition of the state machine in an imperative language, such as C++ or C#, would be much more difficult. Indeed one might think that discriminate unions are equivalent to enumerations, available in C++,Java and C#, but they are much more expressive: an enumeration binds an integer to a ”name”, a discriminate union allows multiple parametric constructors for the same data type (such as in the case of `Wait2` wich takes a float as argument).

One way to implement a discriminate union in an object-orientated imperative language would be through enum and class variables. For example, the following Java-like pseudo-code implements the state machine:

```
public class StateMachine
{
  enum State {Wait1,Wait2,Success,Fail};

  State state;
  double t;

  public void transition(float dt)
  {
    switch(state)
    {
      case Wait1:
        if (is_key_down(key1))
        {
          state = Wait2;
          t = 0.3;
        }
        break;
      case Wait2:
        if (t <= 0.0)
          state = Fail;
        else if (is_key_down(key2))
          state = Success;
        else
```

```
          t = t - dt;
        break;
      default:
        return;
    }
  }

  public void state_function(Light light)
  {
    switch(state)
    {
      case Success:
        light.light_on();
        break;
      default:
        return;
    }
  }
}
```

Furthermore we might need to run concurrently two different tasks and terminate their execution as soon as one of them ends. This leads to an even more complex state machine, since we need to keep executing their code at each frame until one of them terminates and suspend the execution of the other one. As we will see in Section 2.5, this problems are usually solved implementing a scripting language which supports synchronization and waiting primitives.

2.3 General structure of Casanova

The formal grammar definition and semantics of Casanova language will be presented in Section 2.7 for its new version. Here we will give a brief and informal overview of the structure of a Casanova program, in order to better understand the implementation choices.

Casanova 1.0 is implemented in F#, thus it is not compiled, rather interpreted. Casanova uses the *Rule-Script-Drawing* pattern (RSD):

- *rules* manage the continuous aspects of the game.
- *scripts* manage the discrete (or irregular) aspects of the game.
- The *drawing* engine draws the scene after rules and scripts have been run in the current frame.

In the considerations below, we will analyse the mechanism to update rules and run scripts, but we will not consider the draw function which is implemented using an external graphics library and it is not in the scope of this thesis.

A Casanova program is made of a data structure called *world*, which is a container for all the elements and data used in the game. This data structure is unique, i.e. there cannot be more than one world. The elements of the game are called *entities*.

World and entities are implemented in F# using the record data structure. A record is made of a set of fields and possibly methods associated with it, as follows:

```
type R =
  {
    F1 : T1
    F2 : T2
    ...
    Fk : Tk
  }
  member this.M1(a11,a12,...,a1m1}) =
    //body of M1
  member this.M2(a21,a22,...,a2m2) =
    //body of M2
  ...
  member this.Mn(an1,an2,...,anmn) =
```

thus, the world and entities can be defined in the same way

```
type World =
  {
    //world record definition
  }

type Entity1 =
  {
    //entity 1 record definition
  }
...
type EntityN =
  {
    //entity N record definition
  }
```

Rules can be implemented as methods of the record representing the entity (or world). In order to distinguish rules from standard methods, Casanova uses the convention that a rule must have the name of the field it modifies followed by *Rule*. Besides, each rules takes 3 input parameters: `world`, `self`, and `dt`, that are respectively the instance of type `World`, the instance of the entity whose field is being updated, and the discretization step seen in Section 2.1. For example if a rule modifies the field `Fi` of an entity, its definition must be:

```
static member FiRule(world, self, dt) = ...
```

Rules are also associated to a data container implementing the double buffering technique. This technique employs two copies for a value, called v and v'. To avoid race conditions and all the other conflicts arising from the parallel execution of multiple functions, each rule reads from v and writes in v'. Thus, fields associated to a rule must have type `Rule<'a>` or `RuleList<'a>` (in the case of a collection).

Assuming we have a field of type `Rule<int>`, updating a Rule container is done in the following way:

```
static member FiRule(world,self,dt) = self.Fi + 1
```

If we have a `RuleList<int>` then the update must be done using a *List Comprehension* in the following way:

```
static member FiRule(world,self,dt) =
  [for x in self.Fi do
     yield x + 1]
```

The important thing to remember is that the return value of a rule must be of the same type of the field it is updating. So in the case of `Rule<'a>` the return type must be `'a`, in the case of `RuleList<'a>` the return type must be `List<'a>`. That is why, in the case of a `RuleList`, we use a list comprehension to build a list from the current one with the updated values. Note that, in the previous examples, we used neither the `world` nor the `dt` arguments, because we just wanted to increment by 1 the value of the field at each frame, but updating the velocity of an entity requires the following code:

```
static member VelocityRule(world,self,dt) = self.
   Velocity + self.Acceleration * dt
```

Furthermore, the interpreter makes use of *attributes* to distinguish world and entity records from standard F# records. The syntax is the following:

```
type [<CasanovaWorld>] World = ...
type [<CasanovaEntity>] EntityI = ...
```

A Casanova program consists of a set of world and entity definitions. The program entry point (`main` function) takes as argument a world record and, using the interpreter, updates the rules. Besides, in this function is also possible to run coroutines (or scripts) to manage the discrete events of the game (such as inputs, events associated to a specific condition, etc.). In the following sections we will describe how the interpreter updates rules and runs scripts.

2.4 Rule implementation

The updating mechanism for rules makes use of *Reflection* to read the structure of the world and entity definitions and update its fields. Reflection is a set of libraries available in some languages (including F#) which allows to examine and modify the structure and behaviour of a program at runtime. Using reflection we can start from the `World` record and recursively examine its fields and update them if they are marked as rule containers. If one of the field is an atomic type then we have reached a leaf of the recursion tree and we stop the recursion. Otherwise we recursively decompose the non-atomic types and inspect their structure in search of other rules to update. the traversal scheme is shown in Figure 2.1.

We will start examining a naive version of the traversal, which, at each frame, traverses all the game structures and updates the rules accordingly. We then present an optimization of this method. Since both solutions make use of F# *active patterns*, which we will briefly introduce below.

2.4.1 Active Patterns

In functional programming languages we have a more expressive version of the standard `switch` statement. This structure called `match` allowed to test not only a condition on an integer (such as in Java or C++) but also on object constructors such as in *Discriminate Unions*, as shown in Example 2.1.

Example 2.1. Let us suppose we want to implement a weekly calendar, where each day is paired with a string defining a memo. Appointments can be assigned only in a work day (i.e. not Saturday or Sunday). We can define it as a list of Day defined through discriminate unions, which are objects supporting multiple constructors:

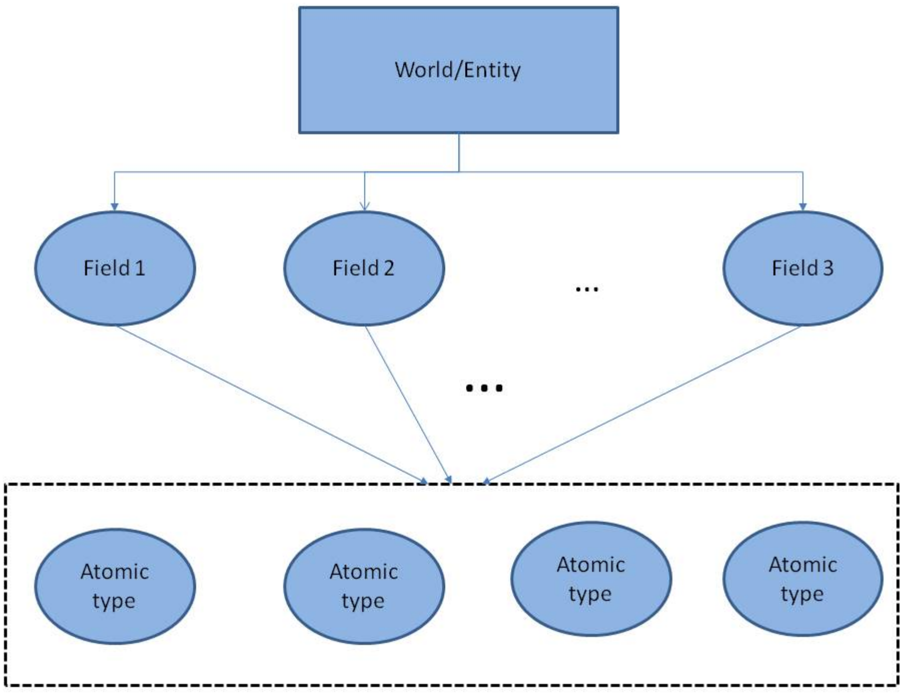

Figure 2.1: Scheme of Casanova traversal

```
type Day =
| Monday of string
| Tuesday of string
| Wednesday of string
| Thursday of string
| Friday of string
| Saturday
| Sunday

type Calendar = Day list
```

we want to implement a function returning the appointment contained in the string contained in the discriminate union. We can use the `match` case to see which day is the one we passed as argument:

```
let get_appointment (day : Day) : string =
  match day with
  | Monday(s)
  | Tuesday(s)
  | Wednesday(s)
  | Thursday(s)
  | Friday(s) -> s
  | Saturday
  | Sunday -> "No appointments during holidays"
```

Active Patterns allow to define named partitions to subdivide input data as in the case of discriminate unions. Example 2.2 shows the use of active patterns to determine if an integer is even or odd.

Example 2.2. Let us suppose we want to implement a function which tells us if a number is even or odd. We could of course return a boolean which is `true` if the number is even, `false` otherwise. A more elegant way of doing this in F# is using active patterns.

```
let (|Even|Odd|) input =
  if n % 2 = 0 then Even else Odd
```

As we can see the function return a value `Even` if the number is even, `Odd` otherwise. We can use a switch statement to print the answer in the following way:

```
let TestNumber input =
  match input with
  | Even -> printfn "%d is even" input
  | Odd -> printfn "%d is odd" input
```

2.4.2 Unoptimized traversal

The traversal function takes as input the `World` type and traverses its fields, and recursively all the fields of the entities, by using reflection. When we encounter a type we traverse its components in turn. When we encounter a Casanova record (a drawable component or a user-defined entity) we invoke `on_casanova_record` function which either executes draw operations or evaluates and executes the rules. This function makes use of active patterns on the input type as described in Section 2.4.1 to hide reflection operations (that are actually executed inside the active pattern).

```
let rec traverse_entity
    ( t_self : Type )
    ( world :'world ) ( self :obj) (dt:float <s>)
    on_casanova_record =
  match t_self with
```

```
| RefType (arg) -> ()
| VarType (arg) | RuleType ( arg) ->
  do traverse_entity arg world ! self dt on_casanova_record
| ListType ( list_arg ) ->
  for x in self do
    do traverse_entity list_arg world x dt
      on_casanova_record
| UnionType ( cases ) ->
  let case , parameters = get_case self
  for parameter , parameter_type in parameters do
    do traverse_entity parameter_type world parameter dt
      on_casanova_record
| CasanovaDrawable ->
  do on_casanova_record world self dt
| CasanovaEntity () ->
  do on_casanova_record world self dt
  for field , field_type in get_record_fields self do
    do traverse_entity field_type world field dt
      on_casanova_record
| _ -> ()
```

The main drawback of this solution is the amount of time taken to execute reflected operations. Indeed, getting a field or a method structure using reflections is computationally very expensive, and the result is the game spends most of the time executing reflected invocations instead of updating the game logic itself. An optimization of this naive function is obtained from the following consideration: the world structure does not change during the program execution, indeed only the values of world or entity fields are changed, but not their definition. Therefore it is useless to keep executing reflected operations at every frame, because their result is always the same. If we can somehow pre-cache their result, we will save a considerable amount of computation time. This optimization can be achieved employing a functional programming technique known as *Continuation passing style*, which will be introduced below.

2.4.3 Continuation passing style

The *Continuation Passing Style* (CPS) employs a continuation function to save the result of the previous recursive step and passing it to the current step to assemble their result. Let us consider, for instance, a function which computes the factorial of n. We recall that

$$n! = \begin{cases} 1 & n = 0, 1 \\ n \cdot (n-1)! & n > 1 \end{cases}$$

Using CPS, when $n = 0, 1$ we pass the continuation function the value 1, which is the result, otherwise we compute the factorial of $n - 1$ and we use the continuation function to assemble the result of the previous recursive step with the current one. This is obtained multiplying the argument of the continuation function by the current number n.

```
let rec factcps n f =
  if (n <= 1) then f 1
  else factcps (n - 1) (fun res -> f n * res)
```

Since we just want the continuation function to return the result, it will be the identity function:

```
let f = fun r -> r
```

The following example shows an execution of the CPS factorial:

Example 2.3. Let us compute the factorial of 5:

```
factcps 5 (fun res -> res) =
factcps 4 (fun res -> (5 * res) -> 5 * res) =
factcps 4 (fun res -> 5 * res) =
factcps 3 (fun res -> 5 * 4 * res) =
factcps 2 (fun res -> 5 * 4 * 3 * res) =
factcps 1 (fun res -> 5 * 4 * 3 * 2 * res) =
5 * 4 * 3 * 2 * 1 = 120
```

This example should clarify the general structure of the Continuation Passing Style: at each recursive step we compute the result of the previous recursive step (that is, $n - 1$), and we pass it to the continuation function which eventually assembles the final result.

2.4.4 Continuation Passing Style cached traversal

From the considerations seen in Section 2.4.3 we can infer that a way cache the reflected operations on the program structures can be implemented using CPS in the following way:

- We build a function which is able to traverse and update the current entity fields (which is the previous recursion step).
- We build a function which updates the current entity and invokes the function built at the previous step.

This can be seen as a CPS function which builds a continuation performing the real exploration of the game world. Since the reflected operations are done within the active pattern, the result will be a function which contains the pre-cached traversal operations, obtained by reflection, and which updates the world entities. The reflected operations are invoked dynamically calling the update function generated at the previous step through CPS. The following program implements the cached traversal described below:[2]

```
let rec traverse_entity
          (t_self:System.Type)
          (k:Ref<'world -> obj -> 'b -> Unit>)
          on_casanova_record =
  match t_self with
  | RefType(arg) -> ()

  | VarType(arg) | RuleType(arg) ->
    let k_aux = ref (fun w s dt -> ())
    do traverse_entity arg k_aux on_casanova_record
        type_predicate
    let k_aux = !k_aux
    let k' = !k

    let value = t_self.GetProperty("Value")
    let value_get = value.GetGetMethod()

    do k := fun world self dt ->
              k' world self dt
              let f = value_get.Invoke(self,[||])
              do k_aux world f dt

  | ListType(list_arg) ->
    let k_aux = ref (fun w s dt -> ())
    do traverse_entity list_arg k_aux on_casanova_record
        type_predicate
    let k_aux = !k_aux
    let k' = !k
    do k := fun world self dt ->
              k' world self dt
              for x in  self do
                do k_aux world x dt
  | UnionType(cases)->
    let tag_reader = precompute_tag_reader(t_self)
    let union_readers =
      [| for case in cases do yield pre_compute_reader(case) |]
```

[2]The ! operator in F# is equivalent to the * operator in C/C++ and it is used to access the value of a pointer.

```
    let parameter_traversals =
      [|
        for case in cases do
          yield
            [
              for parameter in case.Parameters do
                let k_aux = ref (fun w s dt -> ())
                do traverse_entity parameter k_aux
                    on_casanova_record type_predicate
                yield !k_aux
            ]
      |]

    let k' = !k
    do k := fun world self dt ->
              k' world self dt
              let tag = tag_reader self
              let parameters = union_readers.[tag] self
              for p,k in Seq.zip parameters parameter_traversals
                  .[tag] do
                k world p dt

  | CasanovaDrawable ->
    do on_casanova_record t_self k

  | CasanovaEntity(fields) ->
    do on_casanova_record t_self k
    for field in fields do
      let k_aux = ref (fun w s dt -> ())
      do traverse_entity f_type k_aux a type_predicate
      let k_aux = !k_aux
      let k' = !k
      do k := fun world self dt ->
                k' world self dt
                let f = field.Get(self)
                do k_aux world f dt
  | _ -> ()
```

2.5 Coroutine implementation

The discrete component of the game is managed by coroutines, which implements the required synchronization and waiting primitives satisfying the requirements introduced in Section 2.2. Indeed games generally implements a *scripting language* to solve synchronization problems related to discrete events. These languages are usually interpreted and updated by a Finite State Machine. In Casanova we used a different approach since we took advantage of *Monads*, which are a typical functional language construct. They are often used, as in our case, to define *Domain Specific Languages* (DSL). Below we will introduce the Monads and show how they are used to create a DSL with a simple example. Then, we will give a model based on Monads to implement a scripting language.

2.5.1 Monads

In pure functional languages (such as Haskell, F# is a hybrid imperative-functional language) the computation is thought as evaluating expressions. There is not a concept of *state* or *variables*, as in imperative languages. The first issue with this model is that procedural operations such as opening a file or writing an output on a console cannot be treated as expression evaluations[8]. Monads are used to solve this problem in functional programming languages. Indeed, thanks to their properties, it is possible to implement Domain Specific Languages to simulate a change of state.

Formally we defined a Monad as follows:

Definition 2.2. A Monad is a tern formed by:

- A Monadic type `M<'a>`.
- A unary function `return (x:'a):M<'a>`.
- A binary function `bind (m:M<'a>) (f:'a -> M<'b>) -> M<'b>`, usually denoted with the >>= operator.

The unary operator can be seen as a function which takes a value of type `'a` and builds a Monad containing that value. The binary operator can be seen as a function which extracts the value of a Monad, applies the given function to it and returns a new Monad containing the result of the function application.

The Monad must also satisfy the following axioms:

Definition 2.3 (Monad axioms)**.** A Monad is well-defined if it satisfies the following axioms

- `m >>= return` $\equiv$ `m`.
- `return x >>= f` $\equiv$ `f x`.
- `(m >>= f) >>= g` $\equiv$ `m >>= (fun x -> f x >>= g)`

The first axiom states that if you bind a value to a `return` function we get the element itself. The second axiom states that binding a function f to the result of a `return` is equivalent to apply the function itself to the `return` argument itself. The third axiom is the associativity of the `bind` operator.

Optionally a Monad can also have the following properties:

- (*zero*): `mzero >>= f` ≡ `mzero`,`m >>= (fun x -> mzero)` ≡ `mzero`
- (*sum*): `mplus (m1:M<'a>) (m2:M<'a>):M<'a>`

The following example shows how Monads are used to build Domain Specific Languages: let us consider the following imperative code

```
v1 = 5;
v2 = -3;
x = v1;
y = v2;
v1 = x + 1;
z = v1;
return x + y * z;
```

Now we want to use Monads to write the same code in a functional programming language. We recall that, as stated above, functional programming languages are stateless, thus there is no concept of variable. Let us assume for simplicity that our state is made of only two integer locations. A statement in this DSL will be a function which, given as input the current state, applies a function to the state and returns the modified state as result and the result of the function itself. So we define a statament as follows:

```
type Monad<'a * (int * int)> = int * int) -> 'a * (int * int))
```

A Statement is thus a function which takes the current state and returns a value of type `'a` (which is the computation result) and the updated state.

We now define a *step* function, which is a function that, given a statement executes it on the given input:

```
let step (s : Monad<'a * (int * int)>) : 'a * (int * int)) = fun
    m -> s m
```

Now let us implement the unary operator. According to the axioms, the `return` function must take a value of type `'a` and build a monad containing that value. In our case it must build a `Statement` containing that value as result:

```
let Return (x : 'a) = Monad(fun m -> (x,m))
```

The bind operator is implemented using the state passing logic used before: given a statement and a function k, we must pass k the current state, apply k obtaining a new value, and finally build a new statement containing the result of k and the updated state. To get the current state we can use the function `step` defined above.

```
let Bind  (p : Monad<'a * (int * int)>)
      (k : 'a -> Monad<'b * (int * int)>) :
      Monad<'b * (int * int)> =
  fun s -> fun m ->
    let (a,n) = step s m in step(k a) n
let (>>=) p k = Bind p k
```

Let us examine in detail what this function does:

1. We obtain a pair containing the value to modify `a` and the current state `s` using the function `step`.

2. We apply the function `k` to `a`.

3. We apply `step` to the result of the previous step to build the new state.

Now we can implement the functions which allow us to set and get the value of the state. The function to get a variable should return as result the value of the variable itself but should not alter the current state.

```
let v1 = fun (s1,s2) -> (s1,(s1,s2))
let v2 = fun (s1,s2) -> (s2,(s1,s2))
```

The function which sets the value of a variable should not return a value (we use `Unit`), but it alters the current state.

```
let set_v1 n = fun (s1,s2) -> ((),(n,s2))
let set_v2 n = fun (s1,s2) -> ((),(s1,n))
```

At this point we can rewrite the imperative code above using the bind operator and the function to get and set variables, and to return the final value:

```
set_v1 5 >>=
fun _ -> set_v2 -3 >>=
fun _ -> v1 >>=
fun x -> v2 >>=
fun y -> set_v1 (x + 1) >>=
fun _ -> v1 >>=
fun z -> return (x + y * z)
```

The code above does not look very similar to the imperative one. Fortunately F# allows us to map the bind operator into the following syntax: `p >>= (fun x -> ...)` ≡ `let! x = p ...`, thus the code above becomes:

```
let! _ = set_v1 5
let! _ = set_v2 -3
let! x = v1
let! y = v2
let! _ = set_v1 (x + 1)
let! z = v1
return (x + y * z)
```

which is much more similar to the imperative one. Furthermore, one could define a `Combine` function which discards the result of the Bind, that is

```
let Combine (p,k) = Bind(p,fun _ -> k)
```

The syntax mapping for `Combine` is `do! k x`, so the code above becomes

```
do! set_v1 5
do! set_v2 -3
let! x = v1
let! y = v2
do! set_v1 (x + 1)
let! z = v1
return (x + y * z)
```

F# implements monads using a class which contains all the monadic function as methods. A full implementation of the state monad is given below:

```
type Monad<'a * (int * int)> = int * int) -> 'a * (int * int))
let step (s : Monad<'a * (int * int)>) : 'a * (int * int)) = fun
    m -> s m
let v1 = fun (s1,s2) -> (s1,(s1,s2))
let v2 = fun (s1,s2) -> (s2,(s1,s2))
let set_v1 n = fun (s1,s2) -> ((),(n,s2))
let set_v2 n = fun (s1,s2) -> ((),(s1,n))

type StateBuilder() =
  member this.Return (x : 'a) = Monad(fun m -> (x,m))
  member this.Bind
      (p : Monad<'a * (int * int)>)
      (k : 'a -> Monad<'b * (int * int)>) :
      Monad<'b * (int * int)> =
    fun s -> fun m ->
      let (a,n) = step s m in step(k a) n
  member this.Combine (p,k) = this.Bind(p,fun _ -> k)
```

```
let state = StateBuilder()
```

When we require to use monadic code we just write a function building the `StateBuilder`:

```
let m_code(a1,a2,...,an) =
  state{
    //monadic code
  }
```

Note that the state monad can be easily generalized using a generic type `'s` for the state instead of a pair of integer values.

2.5.2 Coroutine monad

From what described in Section 2.5.1, monads are ideal to create Domain Specific Languages. This is our situation, where we have to implement a scripting language to code the discrete behaviour of a game. Our solution comes from the fact that, as seen in Section 2.2, implementing an ad-hoc state machine produces a code which is not flexible, is hard to maintain, and the semantics of the program is lost in a *swithc* (or *match*) structure. Let us define an atomic element characterizing our script: a *step*, which represents the state of a script. The state of the script can be either `Done`, meaning the script has ended its execution, or `Next`. In the latter case the step must embed a continuation marking the next operation to perform. A *script* will be a function which takes no arguments and returns a step:[3]

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

We immediately note the similarity with the state monad: in the state monad with return the result of the application of a function as a value of type `'a`, here the returned value can be either `Done` or `Next`. The continuation (*Next*) can be used to store the world, allowing the script to modify the state of the world, and contains the current state of the script. Note that, with this definition, a script is suspended at every bind and returns a continuation, which is the remaining part to be executed. Furthermore, note that the definition of the script monad is slightly different from the state monad because we do not return a new state at each step. This is due to the fact that the field

[3] The keyword `and` is used in F# to mark mutually recursive type definitions

of the world and entities are *mutable*, thus we can directly modify their values, while in the state monad we had considered the situation of a stateless programming language (such as Haskell).

According to Definition 2.2, we now have to define the `return` and `bind` operators. Returning a value does not require any further computational steps, so we just have to wrap the result of the last step within the `Done` constructor of `Step`.

```
let Return (x:'a) : Script<'a>
  fun () -> Done x
```

When we want to execute a statement, we have to try to execute the first statement: if it is `Done x`, then we perform the binding and continue with the rest of the program. If it is `Next s'` then we cannot invoke the next step yet. We will build a continuation of the script containing the invocation to the binding from where it has stopped at the current step.

```
let rec Bind (p:Script<'a>,k:'a -> Script<'b>) : Script
   <'b> =
  fun () ->
    match p () with
    | Done x -> k x ()
    | Next p' -> Next(bind(p',k))
```

We can now define a script which forces a suspension within the current frame:

```
let yield_ : Script<Unit> =
  fun () -> Next(fun () -> Done ())
```

This script is a function which takes no arguments and return a continuation which does nothing (returning a `Done()` is like doing nothing, since the script will not return a value, nor perform any operations).

Example 2.4. Let us consider the following code snippet (we will use the F# monad syntax as above). Besides assume we have implemented the `Combine` function as in the state monad, that the monad constructor is `co`, and that `s1` and `s2` are other coroutines defined elsewhere:

```
let example =
  co{
    let! a = s1
    if a < 5 then
      do! s2
      return 0
    else
      return a
  }
```

The code above will be mapped (using the bind operator >>= for brevity):

```
s1 >>=  fun a ->
  if a < 5 then
    s2 >>= fun _ -> return 0
  else
    return a
```

Using this monad we can implement a DSL featuring several concurrent operators typical of threaded programs, as shown below.

PARALLEL:

The parallel operator takes two coroutines and waits until both of them has ended their execution, returning their value as a pair.

```
let rec (&&) (s1 : Script<'a>) (s2 : Script<'b>) : Script<'a * '
    b>
  match s1(),s2() with
  | Done x,Done y -> Done(x,y)
  | Next s1',Next s2' -> Next(s1' && s2')
  | Next s1',Done y -> Next(s1' && (return y))
  | Done x, Next s2' -> Next(return x && s2')
```

The parallel operator executes the current statement of the scripts:

- if both are done then it returns `Done(x,y)`.
- If both return a continuation then it returns a continuation containing the invocation of the parallel operator over the remaining part of both scripts.
- if one is done and the other returns a continuation it returns a continuation containing the invocation of the parallel operator on the script which still has to terminate and the `return` of the one which has terminated.

CONCURRENT:

The concurrent operator executes both scripts concurrently and returns the result of the first one to terminate.[4]

```
let rec (||) (s1 : Script<'a>) (s2 : Script<'b>) : Script<Choice
    <'a*'b>> =
  match s1(),s2() with
  | Done x,_ -> Done(Choice1of2 x)
  | _,Done y -> Done(Choice2of2 x)
  | Next s1',Next s2' -> Next(s1' || s2')
```

The code snippet shows that, if one of the scripts is returning a value, than the operator returns that value discarding the other one. If both return a continuation, we call the operator on the continuation of both scripts.

GUARD:

The guard operator takes two scripts and executes the second one only when the first one returns a value.[5]

```
let rec (=>) (s1 : Script<Option<'a>>) (s2 : Script<'b>) :
    Script<'b> =
  co{
    let! x = s1
    match x with
    | None -> return! s1 => s2
    | Some v -> return! (s2 v)
  }
```

The code snippet shows that if the first script does not return a value, then we run the operator again on the same pair of scripts. If the first script returns a value, then we run the second script, passing the result of the first one as argument, and then return its result.

REPEAT:

This operator takes a script and keeps executing it forever.

```
let rec repeat (s:Script<Unit>) : Script<Unit>
  co{
    do! s
    return! (repeat s)
  }
```

[4]The type `Choice<'a,'b>` is a type which can have either type `'a` or `'b`.

[5]The type `Option<'a>` is a discriminate union which can be either `None` (no value) or `Some x` (some value `x`). Also, the keyword `return!` is a F# shortcut for `let! x = v1;return x`.

WAIT:

This script interrupts the current script for the amount of seconds taken as argument.[6]

```
let wait (interval:float32) : Script<Unit> =
  co{
    let! t0 = time
    let rec wait_recursion() =
      script{
          let! t = time
          let dt = (t - t0).TotalSeconds |> float32
          if dt < interval then
            do! yield_
            return! wait_recursion()
      }
    do! wait_recursion()
  }
```

Note that we have to call `yield_` to allow the script to interrupt itself in order to make the timer advance.

CONDITIONAL WAIT:

This script interrupts the current script until the predicate taken as argument is satisfied.

```
let rec wait_condition (p:Unit -> bool) : Script<Unit>
   =
  co{
    if p() then return ()
    else
      do! yield_
      return! wait_condition p
  }
```

As for `wait`, we need to call `yield_` to stop the current script execution, otherwise the script will keep looping within the current frame.

Let us now consider the example introduced in Section 2.2. Expressing such behaviour with our monadic DSL becomes very straightforward:

[6]We assume that we have defined a script `time` elsewhere, which returns the current time.

```
let light_on() =
  co{
    do! wait_condition(fun () -> is_key_down(key1))
    let! x = (wait_condition (fun () -> is_key_down(
       key2)) || (wait 0.3f))
    match x with
    | Choice1of2 _ ->
      do light_on()
      return ()
    | Choice2of2 _ -> return ()
  }
```

The script waits until `key1` is pressed and then invokes a concurrent operator over a script which waits for `key2` to be pressed and a script which waits for 0.3 seconds. If the returned value belongs to the first script it turns the light on, otherwise it does nothing.

2.6 From Casanova 1.0 to Casanova 2.0

Casanova 1.0 has been evaluated deeply by making students build some games with it and by building several samples on our own. The language has revealed to be more synthetic (less lines of code for the same sample) and faster than scripts made in Python, Lua and C#. In particular Python and Lua suffer a performance drop, possibly due to dynamic lookups to access the game state [12]. This loss of performance is, of course, accentuated in actual games, where the game state is a very complex structure. Tests on students showed that Casanova is appreciated for its expressiveness and easiness to manipulate complex entities. In particular the rule system has shown to be able to express complex dynamics in a very synthetic and straightforward way. Besides, the drawing part does not require deep knowledge of geometry and linear algebra, as in the case of programming a graphics engine from scratch, so even 1st year students were able to implement their samples and draw them with almost no mathematical knowledge about transformation matrices, vector operations, etc.

Casanova 1.0 however suffers of some flaws:

- A loss of performance in scripts due to the monadic implementation.
- An overlapping semantics between rules and scripts.

- Advanced users want easier integration with better libraries.
- Advanced users want better code reuse mechanisms in line with their knowledge of Object-Oriented languages.

The first point is due to the fact that, as we explained above, monads require a heavy usage of anonymous functions or lambda abstractions (called `fun` in F#). The F# compiler converts these functions into classes, which contains an *invoke* method to run the actual body of the function. Of course this implies that, for each bind, the program must access the object layout and then access the dispatch vector to run the method. This point can be solved implementing an actual compiler for our language to get rid of the monadic class.

The second point comes from the student evaluation: we noticed that the rules and scripts are used almost in an exclusive way by each test subject. Many students use scripts just when they have to code behaviours depending on a user input, but in several cases even discrete time events where modelled as rules implementing an internal counter.

The main challenge of designing Casanova 2.0 will be focused around defining a new language structure unifying rules and scripts, and add Object-Oriented features to the new version of the language.In Section 2.7 we will redesign the language to reflect the feedback we received from our test subjects, in Chapter 3 we will describe the compiler architecture.

2.7 Casanova 2.0 language definition

We will define our new languages covering the following steps:

1. Removing the separation between rules and scripts.
2. Adding Object-Oriented code structures.
3. Supporting declarative SQL-style queries on entities.

2.7.1 Unifying rules and coroutines

Unifying rules with scripts requires to take the expressive power of Casanova and reproduce it with a smaller set of primitives. Besides the unification process does not simply consist in taking what we already have in Casanova and putting it inside rules, rather in reducing it inside rules. This is due to the fact that rules appear to be the most intuitive and expressive construct of Casanova.

Our new design will allow:

- rules over multiple fields.
- interruptible rules as scripts were in Casanova 1.0.
- overlapping rules: multiple rules will be able to update the same fields.

This means that if a single entity can be updated according to multiple logics, then these logics are stored separately.

Let us consider the case of a physical body without rotation. This body has a position and a velocity updated according to the approximation of the kinematics differential equations shown in Section 2.1. A physical body also tracks colliders, which are those bodies colliding with the current entity. When a collider is found, then a new rule updating both position and velocity is run. The following code snippet describes this behaviour:[7]

```
entity Body =
  {
    Position  : Vector2<m>
    Velocity  : Vector<m/s>
    Radius    : float32<m>
    ref Colliders : List<Body>

    rule Position = yield self.Position + self.Velocity * dt
    rule Velocity = yield self.Velocity + G * dt
    rule Colliders =
      let cs =
        from b <- *
        where b <> self && collides(self,b)
        select b
      yield cs
    rule Position,Velocity =
      wait_until (Colliders.Length > 0)
      //compute the dynamics of the collision storing it in P
         and V...
      yield P,V
  }
```

In the code above the statement `yield` is used to tell the rule to update the field `Colliders` (it has a similar semantics as a `return` statement in imperative languages).

[7]Note that, for brevity, we have omitted the fact that Casanova supports units of measure associated to constant values. A unit of measure is specified in this way: `let v = l<m>`, where `l` is one of the constant literals available in F# and `m` is a defined unit of measure. Also, note that the keyword `ref` marks a field which is not touched during the world traverse, but just defined in the current entity to be readable.

The new features can immediately be seen in the previous example: we have a SQL-like statement to perform a query over an entity [8].We have a *joint rule* which modifies two fields at the same time. The same rule is also an *interruptible rule* since it has a `wait_until` statement in its body.

2.7.2 Object orientation

A first feature typical of Object-Oriented languages available in Casanova 2.0 is *inheritance*. Let us suppose we want to build a game with ships which behaves like the physical body described in Section 2.7.1. We can simply inherit the `Body` entity and just define the code to update its additional fields[9]

```
entity Ship =
  {
    inherit Body
    Sprite : Sprite

    rule Sprite.Position = self.Body.Position * 1<pixel
      /m>
  }
```

Let us suppose we want to add a new entity `Shipyard`, which behaves like a physical body as well. We will write

```
entity Shipyard =
  {
    inherit Body
    ...
  }
```

Thanks to the inheritance, we can reuse the code defined in `Body` to simulate the same behaviour also for `Ship` and `Shipyard`. Note that Casanova 2.0 supports constructor for entities using the keyword `create`:

```
entity Body =
  {
    Position : Vector2<m>
    Velocity : Vector<m/s>
```

[8]Actually a similar statement was available in Casanova 1.0, although it was not part of the language itself, in the sense it was not part of the syntax, since the language uses F# syntax and such statement is not available. For further information see [?]

[9]A `Sprite` is a data structure used to draw entities by the graphics engine, which is not in the scope of this work.

```
    Radius    : float32<m>
    ref Colliders : List<Body>

    rule Position = yield self.Position + self.Velocity * dt
    rule Velocity = yield self.Velocity + G * dt
    rule Colliders =
      let cs =
        from b <- *
        where b <> self && collides(self,b)
        select b
      yield cs
    rule Position,Velocity =
      wait_until (Colliders.Length > 0)
      //compute the dynamics of the collision storing it in P
         and V...
      yield P,V

    create(x,y,r) =
      {
        Position = Vector2.Create(x,y)
        Velocity = Vector2.Zero
        Radius   = r
        Colliders = []
      }
  }
```

A constructor can also be used with inheritance:

```
entity Ship =
  {
    inherit Body(x,y,r)
    Sprite : Sprite

    rule Sprite.Position = self.Body.Position * 1<pixel/m>

    create(x,y,r,texture) =
      {
        Sprite.Create(//arguments of sprite creation, including
           texture...)
      }

  }
```

The language supports multiple inheritance, since for example an entity might be *physical,damageable,drawable,damager* and many other common components which implement their own code which can be reused.

Note that rules are updated in cascade, this means that, when inheriting an entity, if both the inherited entity and the inheriting one have a rule over the same field, this is updated twice. For instance, consider the following code snippet:

```
entity A() =
  {
    Counter : int
    rule Counter == yield Counter + 1

    create() =
      {
        Counter = 0
      }
  }

entity B() =
  {
    inherit A()

    rule A.Counter = yield A.Counter + 2
  }
```

In this example `Counter` is updated twice and incremented by 3, since the rule in `A` updates it by 1, and then, in cascade, the rule in `B` is called to update the same field by 2.

A rule can be marked as `virtual`, meaning that inheritors can override the rule, in order to avoid the phenomenon described above. Let us consider an alternative version of the previous code snippet:

```
entity A() =
  {
    Counter : int
    virtual rule Counter = yield Counter + 1

    create() =
      {
        Counter = 0
      }
  }
entity B() =
  {
    inherit A()

    rule A.Counter = yield A.Counter + 2
  }
```

In this case `Counter` is incremented only by 2 because `B` overrides the rule in `A`.

Finally a rule can be marked as `abstract`, meaning that inheritors must provide an implementation of that rule:

```
entity A() =
  {
    Counter : int
    abstract rule Counter

    create() =
      {
        Counter = 0
      }
  }
entity B() =
  {
    inherit A()

    rule A.Counter = yield A.Counter + 1
  }
```

Let us consider the following example, where an entity inherits more than one entity, each one having its `Position` field, which must be updated.

```
inherit PhysicsBody
inherit DamageGiver
inherit DamageTaker

rule DamageGiver.Position = yield PhysicalBody.Position
rule DamageTaker.Position = yield PhysicalBody.Position
```

In this situation, the inheritor must provide a rule to update both fields of the other inherited entities. This situation is not only error-prone but also inefficient because it requires needless copies of the same field.

To avoid this situation a field can be declared as `abstract`, this means that an entity declares the field but does not store it. In this way, the inherited entities in the snippet above can declare `Position` as `abstract`, and having the ”real” field stored in the inheritor.

An additional method is that of declaring a field `virtual`. A virtual field, much like an abstract one, may be overridden. When this happens, then the two declarations are essentially merged into a single concrete field shared by both entities. If, for example, we declared the `Position` field inside `PhysicsBody` as virtual, then we could inherit all the base entities with the same code above. If we need to inherit just the `PhysicsBody` though, then we can skip the redefinition of the `Position` field:

2.7.3 Language syntax and semantics

The syntax of the language (here presented in Backus-Naur form) is rather short. It allows the declaration of entities as simple functional types (records, tuples, lists, or unions). Records may have fields and may inherit from other records. Rules contain expressions which have the typical shape of functional expressions, augmented with `wait`, `yield`, and queries on lists.

```
<Program > ::=        <moduleStatement>
               {< openStatement >}
<worldDecl > ::=    world id "=" <worldOrEntityDecl>
<entityDecl> ::=    entity id "=" <worldOrEntityDecl>
<worldOrEntityDecl> ::=
  "{"
    [inherit id {"," id}]
    <entityBlock>
  "}"
<entityBlock> ::= {<fieldDecl>} {<ruleDecl>} {<constructor>}
<formal>     ::= id [":" <type>] {"," <formals>}
<fieldDecl>   ::= [virtual] id [":" type] ["=" <expr>] |
    abstract id [":" <type>]
<ruleDecl>     ::=
  ([override | virtual] rule id {"," id} "=" <expr> | abstract
      rule id {"," id}) "="
  <expr>
<constructor> ::= create "=" [<letBindings>] "{" <createBody> "}
    "
<letBindings> ::= let id "=" <expr>
<createBody>  ::= id "=" <expr>
<type>       ::= int ["<" measureType ">"]
             | boolean
             | float32 ["<" measureType ">"]
             | Vector2 ["<" measureType ">"]
             | Vector3 ["<" measureType ">"]
             | string
             | char
             | list "<" type ">"
             | <type> "[" "]"
             | <generic>
             | id
<generic> ::= "'" id
<measureType> ::=    "1"
             | id
             | <measureType> "*" <measureType>
             | <measureType> "/" <measureType>
 <expr>        ::=  (*typical F# expressions such as let, while,
      etc.)
                    | <queryExpr>
                    | wait | yield | <arithExpr> | <boolExpr> | <
```

```
                literal>
<queryExpr>  ::= from id [",", id] "<-" id where <boolExpr>
    select <expr>
```

The semantics of Casanova are mostly rewrite-based[13] [11], meaning that the current game world is transformed into another one with different values for its fields and different expressions for its rules. The two exceptions to the rewrite-based semantics are rendering and input. The semantics of rendering assume that after every tick of the game loop, the drawable entities are drawn to the screen, whereas the semantics of input assume that the evaluation of the input functions such as `IsKeyDown` or `GetMousePosition` looks up the latest state of the input buffers.

Let us consider the following example as an instance of rewrite-based semantics:[10]

```
world World =
{
  X : float <m>
  V : float <m/s>
  rule X' = X + V * dt
  rule V' =
    wait (IsKeyDown (Keys.Space))
    yield 1<m/s>
    wait 2<s>
    yield 0<m/s>
}
```

The game world needs to track information about the values of its field, but also the current execution state of the rules over those fields. In particular Rule `V'` needs to track at which point its execution it is now, i.e. if it is at the beginning or during the second `wait` statement. This implies the world will contain 3 values, two for the fields and one for the rule.

Let us consider a long game interval, with $dt = 1s$, the state of the game world would be

```
W = {X = 0; V = 0; rule V' = wait(IsKeyDown(Keys.Space))...}
```

Let us assume now that `Space` is pressed at some point. The state will be the following:

```
W = {X = 1; V = 1; rule V' = wait 2.0 ...}
```

[10]Note that we used a different field name for the rule assignment just for the purpose of distinguish rules from fields by name in what follows.

After $dt = 1s$ has passed, we will obtain the following state:

```
W = {X = 2; V = 1; rule V' = wait 1.0 ...}
```

When the timer has exhausted Rule `V'` goes back to the beginning and `V` is zeroed again thanks to the `yield` statement returning 0.

More formally let us define a transformation $[[\bullet]]$ from entities into entities applying rules and replacing field values. The transformation also applies itself recursively to all fields. We will assume, as above, that `world` is the game world and `dt` is the amount of time elapsed from the last tick. Besides, let us define the semantics for a single rule as a transformation $\langle\bullet\rangle$ which maps expressions in a pair (e', v') where e' is the new body of the rule (an expression as well), and v' is the new value of the field.

```
[[type T = {f1 : F1;f2 : f2; ...; fn : Fn;
  r1 = R1;...;rm = Rm}]](w : World,t : T, dt : float<s>)
∀f_i, r_j : r_j = f_i,   f_i, r_j = ⟨t.R_j⟩ (w, t, dt, [[t.f_i]](w, t, dt), R_j)
∀f_i : ∄r_j, r_j = f_i   f_i = [[t.f_i]](w, t, dt)
```

The transformation above takes each field with a corresponding rule and produces a new value for both the field and the rule itself. For all fields, whether or not they have been transformed by the rule, we then reapply the original semantics. When the evaluation of the rule body reaches the end, then the semantics returns the previous value of the field f, as the new value, and the initial body of the rule as the new body.

```
⟨ϵ⟩(w,t,dt,f,R) = f,R
```

For all other types, such as lists, discriminate unions, tuples, the operator $[[\bullet]]$ is simply applied respectively to all its elements, union cases, components.

For most of terms the semantics are the same of pure functional programming languages:

```
⟨ let x = e1 in e2 ⟩(w,t,dt,f,R) =
  ⟨ e2[x → e1[world → w,self → t,dt → dt]] ⟩(w,t,dt,f,R)
⟨ if true then e1 else e2 ⟩(w,t,dt,f,R) =
  ⟨ e1 ⟩(w,t,dt,f,R)
⟨ if false then e1 else e2 ⟩(w,t,dt,f,R) =
  ⟨ e2 ⟩(w,t,dt,f,R)
...
```

The terms that require special care are those involving waiting or yielding:

```
⟨ wait 0<s>;k ⟩(w,t,dt,f,R) = ⟨ k ⟩(w,t,dt,f,R)
⟨ wait w_t;k ⟩(w,t,dt,f,R) = ⟨ wait(w_t − dt);k⟩(w,t,dt,f,R)
⟨ yield f';k⟩(w,t,dt,f,R) = f',k
```

The first rule is the basic step of the next recursive rule . A `wait` of 0 seconds means evaluating immediately the continuation k of the rule.

The second rule states that a `wait` of w_t seconds returns the set of fields followed by a new code for the rule where the wait time is now $w_t - dt$.

2.8 Coroutine operators in Casanova 2.0

By removing coroutines from Casanova 1.0 we lose all the concurrency operators defined in Section 2.5.2. Thus, it would appear that Casanova 2.0 is less expressive than Casanova 1.0, since we cannot make use of functions which implement parallel and concurrent behaviours as for coroutines. Below we try to implement some of the operators defined for the coroutine monad to show that, using interruptible rules, we can achieve the same expressiveness of coroutines.

WAIT and CONDITIONAL WAIT

This two operators are already implemented as language primitives in Casanova 2.0, and they are identical to their counterparts in the coroutine monad.

PARALLEL

The implementation of the parallel operator is not straightforward. We recall that this operator runs two coroutines in parallel and waits until both return a result. We can model a coroutine as an entity which contains two fields: an input parameter and one containing its state. The coroutine will be constructed setting its state to `Running` which, when it has to return a result, will be changed to `Done`. Its only rule will act on the result and will contain the code which the coroutine has to execute. The following code gives the full implementation:

```
entity State<'T> =
| Done of 'T
| Running

entity Coroutine =
  {
    Input   : Option<'T1>
    State : State<'T2>
  }

  rule Input,State =
    wait_until Input = Some input
    //execute coroutine code
    yield None,Done(x)

  Create(input : 'T1) =
    {
      Input = Some input
      State = Running
    }
```

As we can see, the coroutine waits until it has an input, it executes the code and then sets its input to `None` to stop the rule, and changes its state to `Done`.

We now define a *coordinator entity*. This entity will contain two fields referencing the coroutines, and a field containing the result of the parallel operator. The result is an option since it has a value only when the result of the parallel is produced . This entity constructs the two coroutine entities and runs them, waiting until they both produces a result.

```
entity E =
  {
    C1  : Option<Coroutine>
    C2  : Option<Coroutine>
    V : Option<'a * 'b>
  }

  rule C1,C2,V =
    wait_until C1 = Some(c1) && C2 = Some(c2)
    wait_until C1.State = Done(x1) && C2.State = Done(x2)
    yield None,None,Some (x1,x2)

  Create(input1,input2) =
  {
    C1 = Coroutine.Create(input1)
    C2 = Coroutine.Create(input2)
    V = None
  }
```

The rule will wait until the coroutine have been created. Then it waits until both produces a result. When both results are ready it sets the value to the pair containing both coroutine results.

CONCURRENT

The structure of the concurrent operator is similar, but we have to take only the result of the first coroutine which terminates. The coordinator entity definition is the following:

```
entity E =
  {
    C1   : Option<Coroutine>
    C2   : Option<Coroutine>
    V : Option<'T>
  }

  rule C1,C2,V =
    wait_until C1 = Some(c1) && C2 = Some(c2)
    wait_until C1.State = Done(x1)  || C2.State = Done(x2)
    if C1.State = Done(x1) then
      yield None,None,Some x1
    else
      yield None,None,Some x2

  Create(input1 ,input2) =
  {
    C1 = Coroutine.Create(input1)
    C2 = Coroutine.Create(input2)
    V = None
  }
```

The coordinator entity waits, as usual, that the coroutine entities are created. Then waits until either one of them returns a value. If the first one has returned then it sets to `None` both coroutine entities and yields the returned value. In this way the reference to the other coroutine entity will be set to `null` and the execution of the rule of the second coroutine entity will be stopped. If the second one has returned then both are set to `null` and we return the result of the second one.

GUARD

The guard operator requires to wait for the first coroutine entity to return. The returned value of the first coroutine entity will be passed as input to the second one, which eventually will return its own result.

We have to change the constructor of the coordinator entity, so that it will only construct the first coroutine entity. The second one will be constructed after the first one returns. The definition of the coordinator entity in this case is the following[11]:

```
entity E =
  {
    C1   : Option<Coroutine>
```

[11]The statement `yield FieldName <- value` is a variant of the `yield` statement which allows to yield on just a subset of the fields modified by the rule.

```
    C2   : Option<Coroutine>
    V : Option<'b>
  }

  rule C1,C2,V =
    wait_until C1 = Some(c1)
    wait_until C1.State = Done(x1)
    yield C2 <- Some (Coroutine.Create(x1))
    wait_until C2.State = Done(x2)
    yield None,None,Some x2

  Create(input1) =
  {
    C1 = Coroutine.Create(input1)
    C2 = None
    V = None
  }
```

The coordinator rule waits until the first coroutine entity is created. Then it waits until the first coroutine entity returns a value. Then the second coroutine entity is created and the result of the first one is passed as argument. Finally it waits until the second coroutine entity returns and yields the result in the corresponding field in the coordinator entity.

From the considerations above we showed that the new interruptible rules introduced in Casanova 2.0 are no less expressive than coroutines, since we can implement the concurrency operators we had previously defined.

Chapter 3

Casanova compiler

In this chapter we will examine the architecture of Casanova 2.0 compiler. This compiler is widely based on the open-source version of F# compilers. This choice comes from the fact that we want to maintain compatibility among Casanova compiled programs and .NET libraries. This gives us the following advantages:

- We can write external libraries supporting our Casanova program, such as file readers as we will see in Section 4.1.
- We can implement interoperation between third-party .NET libraries, such as external engines as Unity 3D, XNA or MonoGame.
- We can use F# compiler back-end to produce .NET bytecode without writing our type checking rules and code generation procedures.

Casanova compiler will just implement the front-end, that is the lexer,parser and Abstract Syntax Tree (AST), while it will delegate the code checking and generation to F# type checker and back end. This can be achieved by translating Casanova AST into F# AST, using a set of mapping functions. The mapping is almost one to one since Casanova AST, is just a pruned and slightly modified version of F# AST, and it will not be covered deeply by this thesis, as it is just a mere translation from Casanova AST types into F# AST types.

A scheme of Casanova compiler is shown in Figure 3.1. Our compiler will take as input a Casanova source file (CNV file), and it performs the Lexing/Parsing operations defined according to the syntax rules given in Section 2.7.3. This will produce Casanova AST. The AST is then modified to reflect shadowing, as described in Section 3.4, and then the rules state machine is generated. This part will be covered by Section 2.2.

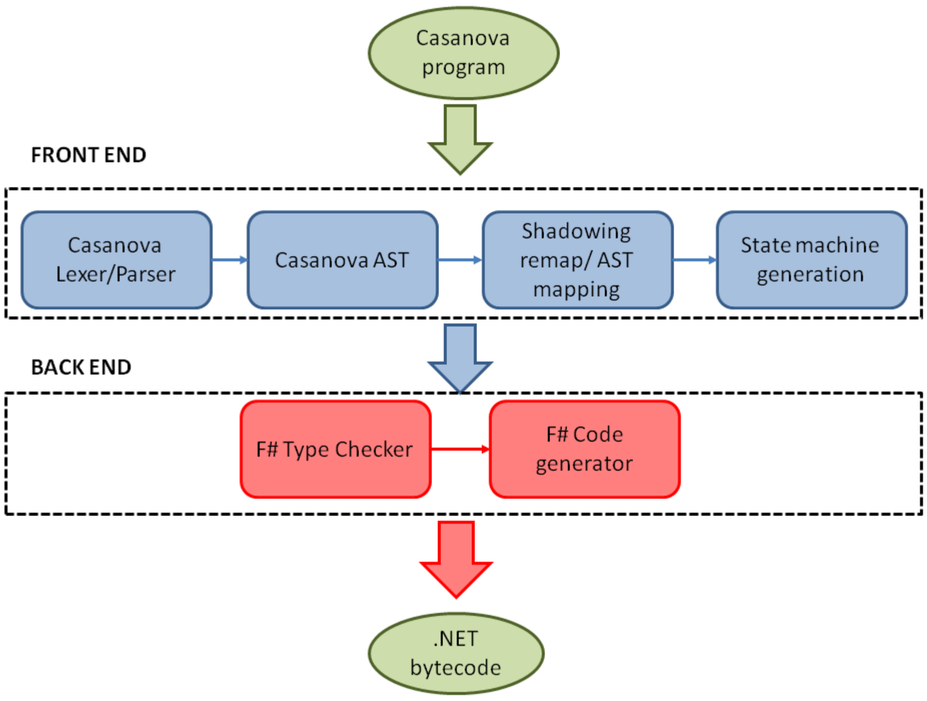

Figure 3.1: Casanova Compiler architecture

3.1 Casanova lexer

Casanova lexer is a modified version of F# lexer. The keywords of Casanova have been added to this lexer in order to implement the new control structures of the language, such as `rule`,`from`,`where`,`select`,`world`,`entity`. An interesting feature of this lexer is the usage of a stack to detect the beginning and end of a block. F# does not require block delimiters such as curly brackets in Java/C++/C#, rather it uses indentation as a syntax rule to define a block. This, unfortunately, makes F# a context-dependent language. The context is the position of the token in the file.

When running the lexer, the compiler maintains a stack where it pushes support tokens at the current token t context, i.e. its row and column position, into the stack. Whenever it encounters a new token t' it tries to close the current block if t' context is the same of t context, i.e. if they are aligned, pushing a *block end* token. Otherwise it pushes an *end coming soon* and a *block begin* token to signal that we are waiting to close the block in t and we are opening a new block in t'.

Example 3.1. Consider the following code snippet

```
let f x y =
  let a = f2 x
  let b = a**2 + (f3 y)
  a + 2 * b + (x * y)
...// other code
```

In the following scheme we represent the state of the stack after analysing the statement to the left.

```
let f x y ->
  block_begin
let a ->
  block_begin
  end_coming_soon
  block_begin
let b ->
  block_begin
  end_coming_soon
  block_begin
  block_end
  block_begin
a + 2 * b + (x * y) ->
  block_begin
  end_coming_soon
  block_begin
  block_end
  block_begin
  block_end
  block_end
```

The inner let bindings are closed immediately because they are in the same context. The outer let is closed at the end, but in its body we push a `end_coming_soon` to signal that the following block is in another context. The `end_coming_soon` is then matched with the last `block_end`.

3.2 Casanova parser

Casanova parser is again a modified (and pruned) version of F# parser. The parser is defined and generated using FSYacc, which is a parser generator for F#. In this generator each syntax rule has a name followed by a discriminated union of syntax rules, each one consisting of a sequence of tokens and/or productions. Each syntax rule is followed by F# code used to instantiate the corresponding structure in Casanova AST. Below we will map the syntax definition given in Section 2.7.3 into FSYacc definitions. In what follows we will omit the code used to generate the AST and focus only on the grammar productions. Besides we will examine only the syntax rules related to Casanova structures, omitting those of F#.[1]

WORLD AND ENTITIES:

```
type_keyword tyconDefn tyconDefnList

TYPE_COMING_SOON type_keyword { }
   | TYPE_IS_HERE { }
   | TYPE { }
   | WORLD { }
   | ENTITY { }

tyconDefn:
   | typeNameInfo EQUALS tyconDefnRhsBlock

tyconDefnRhsBlock:
| OBLOCKBEGIN  cnvTyconDefnRhs opt_OBLOCKSEP opt_classDefn
    oblockend opt_classDefn

cnvTyconDefnRhs:
   | tyconDefnOrSpfnCnvRepr

tyconDefnOrSpfnCnvRepr:
   | opt_declVisibility braceFieldOrRuleDeclList:

tyconDefnOrSpfnCnvRepr:
   | opt_declVisibility braceFieldOrRuleDeclList

braceFieldOrRuleDeclList:
   | LBRACE  fieldOrRuleList rbrace
```

World and entities start with a `type_keyword`, which can be a token `world,entity` or `type`. It is then followed by one or more type definition (the third production is used to define recursive types with the keyword `and`). Each type definition is made of a type name, followed by a `=` token and type definition block. Each block is enclosed by curly brackets and contains a list of fields,rules, and a type constructor. The body of an entity definition is the following:

[1]Tokens starting with OBLOCK are the auxiliary tokens described in Section 3.1.

```
fieldOrRule:

/* SynM4emberDefn.Member */
   | opt_declVisibility RULE memberCore  opt_ODECLEND

   | opt_declVisibility CREATE
      createMemberCoreWithParams  opt_ODECLEND


   | opt_declVisibility CREATE createMemberCore
      opt_ODECLEND

   | OLET opt_declVisibility  cnvFieldInitCore
      opt_ODECLEND

   | fieldDecl
```

FIELDS:

```
fieldDecl:
  | opt_ref ident COLON  typ
```

A field can start with an optional reference attribute, which tells if the field is a reference, i.e. it is not updated by the game logic when updating the entity fields, or drawn. Each field has an ID (name) followed by a : and a type name.

RULES:

The syntax definition for rules is the following:

```
fieldOrRule:
  | opt_declVisibility RULE memberCore  opt_ODECLEND
```

A rule begins with optional visibility parameters (`virtual,abstract`), a `rule` keyword and then a member core definition. A member core is a modified version of F# method core, without a method name. It accepts a list of parameters (whose type is optionally specified), followed by a `=` token and a *typed sequential expression block*. This block is the actual body of a rule, and it will be examined below.

CONSTRUCTORS:

The syntax definition for a constructor is the following:

```
fieldOrRule:
  | opt_declVisibility CREATE
      createMemberCoreWithParams  opt_ODECLEND
  | opt_declVisibility CREATE createMemberCore
      opt_ODECLEND
```

The syntax accepts two version of the constructor: one with parameters and another with no parameters.

```
opt_inline createBindingPattern
    opt_topReturnTypeWithTypeConstraints EQUALS
    typedSeqExprBlock
opt_inline opt_topReturnTypeWithTypeConstraints EQUALS
    typedSeqExprBlock
```

The first production accepts arguments for the constructor (`createBindingPattern`) followed by a `=` token and a typed sequential expression block. The second one does not require any arguments.

SEQUENTIAL EXPRESSIONS:

Sequential expressions are the core of rules body. They contains all the F# structures (`let,while,if,for,...`). For brevity we will examine only syntax rules for new Casanova statements, skipping those available in F#.

The first statement is the `wait` statement, whose production is the following:[2]

```
| OWAIT typedSeqExprBlock hardwhiteDefnBindingsTerminator
```

The statement starts with a `wait` token and it is followed by a typed expression. Clearly the typed expression must return a floating point value, but this constraint is checked later by the F# type checker.

The second statement is the `wait_until` statement. The syntax is the same of the `wait` statement, except for the starting keyword:

```
| OWAIT_UNTIL typedSeqExprBlock hardwhiteDefnBindingsTerminator
```

Another statement is the `yield` statement, whose syntax is the following:

```
| OYIELD typedSeqExprBlock hardwhiteDefnBindingsTerminator
```

This statement starts with the `yield` keyword and it is followed by a sequential expression. Again, the grammar allows to write any sequential expression, without checking the returned type, because it will be taken care of by the F# type checker after the mapping procedure.

[2]In the following productions all the tokens OWAIT,OWAIT_UNTIL,etc. are auxiliary tokens used by the lexer as described in Section 3.1.

The `from-where-select` statement is more complex:

```
| FROM fromExprCondition fromExprWhere fromExprSelect

fromExprCondition:
  | fromTupleExpr BAR fromLoopBinder

  | fromLoopBinder

  | fromLoopBinder BAR fromLoopBinder

fromLoopBinder:
  | parenPattern LARROW declExpr
  | parenPattern LARROW rangeSequenceExpr
```

The `from` section is defined writing an id followed by a `<-` operator and an id of a field. If multiple selections are performed, they must be separated with the `|` separator. The right side of `<-` operator might also be a *ranged expression*, that is an interval of integer values. The syntax for the `where` section is the following:

```
| OWHERE OBLOCKBEGIN typedSeqExpr oblockend
```

It starts with a `where` keyword and then it accepts a typed sequential expression. Again, checking that the type is boolean is delegated to F# type checker. The syntax for the `select` section is the following:

```
| OSELECT OBLOCKBEGIN typedSeqExpr oblockend
```

It starts with a `select` keyword and then it accepts a typed sequential expression.

3.3 State machine

In this section we will present the implementation of our new design of rules. We recall that our idea is merging the features of old Casanova scripts into rules, so that we can use time primitives within a rule body. Our solution was to implement a general and compiled version of the state machine introduced in Section 2.2, which is non-trivial[18].

A first naive attempt of implementing a state machine was to reduce each statement in a rule to a state in the state machine, and increase the state at each evaluation of the rule to advance to the next statement. This solution has, however, a major flaw, which is shown by this code snippet:

```
rule X =
  let $x_1$ = $v_1$
  let $x_2$ = $v_2$
  ...
  let $x_n$ = $v_n$
  yield $\sum_{i=1}^{n} v_i$
```

This code is mapped to a state machine which has a state for each of the `let` statements plus one for the `yield` statement. This means that the rule is evaluated $n + 1$ times (once for each statement), thus the body of the rule takes $(n+1) \cdot dt$ seconds to be evaluated. If $dt = 1/60$ seconds and $n = 59$ the body of the rule takes 1 second to be fully evaluated, while clearly it should take just dt seconds because only the `yield` statement alters the entity state.

A better solution is implementing the state machine as a set of code blocks organized in a tree structure. A block has a *type*, a direct link to its *parent*, and a *state*. The body of a rule is a *root block*, which can have several children, depending on the time primitives and the statement defined within it. In the current version of the compiler the block kinds are the following:

- Wait
- Wait_Until
- Yield
- Foreach
- Query
- While
- Let
- If
- Then
- Else
- Atomic
- Body

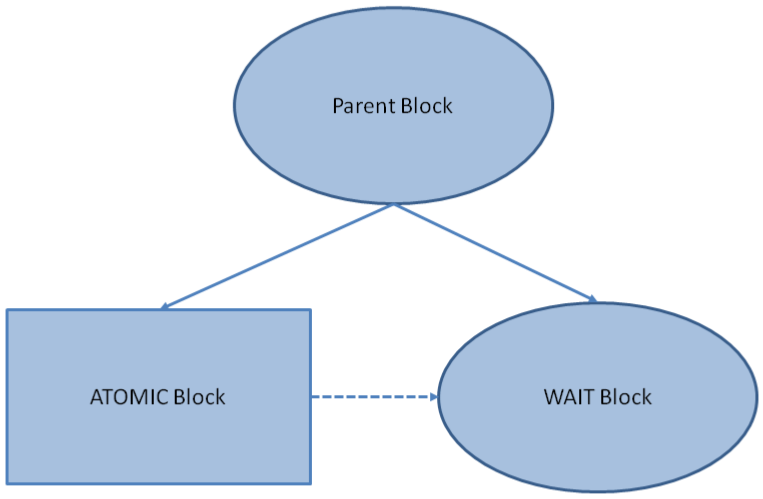

Figure 3.2: Wait block expression

Atomic blocks are made of uninterruptible statements (such as in the previous example) and are executed atomically within the current frame. The other blocks are generated according to their own rule. We will discuss separately the generation of each type of block and their related state machine. In what follows we will use a high-level pseudocode, but in the actual implementation of the compiler each auxiliary data structure is translated directly into F# AST. This choice has been made in order to help the reader, who has not a deep knowledge of the data structure used by F# AST, to better understand the logic behind the state machine and to provide him with readable code (putting a constructor which takes 20 arguments into the code would indeed make it unreadable).

3.3.1 Wait state machine

The block for a `wait` statement has no children and it is obtained by building an *atomic* block containing all the non-interruptible statements preceding the `wait` and the *wait block* itself. A schematic representation is shown in Figure 3.2.

The state machine for this statement is very simple, having just one state.

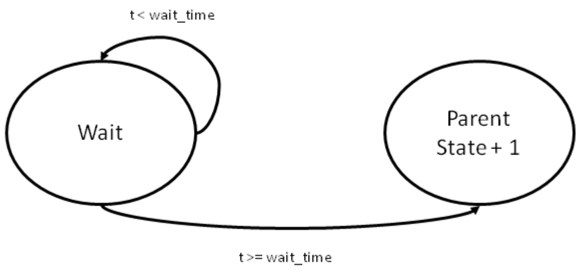

Figure 3.3: Wait state machine

We assume that, during the block generation, we had added a field to the current entity as a timer for the current statement. We then compare the current timer value with the argument of the `wait` statement. If it is lower we increment the timer by *dt*; at the next step the rule will re-evaluate the same condition with the updated timer. Otherwise the timer has expired, thus we reset the timer (because when we start re-evaluating the entire body of the rule we want the timer to start from 0 again), and we increase the state of the parent block (which might be a root block or another kind of block),re-evaluating the rule thereafter. The pseudocode below implements what explained above.

```
match parent_state with
...
| wait(wait_time) ->
  if tk < wait_time then
    tk := tk + 1
  else
    reset_timer(tk)
    inc(parent_state)
    reevaluate_rule()
| next_state ->
  //<- after running the else block the current state is this
```

Figure 3.3 shows a state machine diagram for the `wait` statement.

3.3.2 Wait Until state machine

The `wait_until` statement block is identical to the `wait` block. The state machine is only different in the fact we check if the condition passed as argument to the `wait_until` statement is true. If it is satisfied we increment the parent state and we re-evaluate the rule,otherwise we just do nothing (we return unit at the end of the match case). The pseudocode below implements

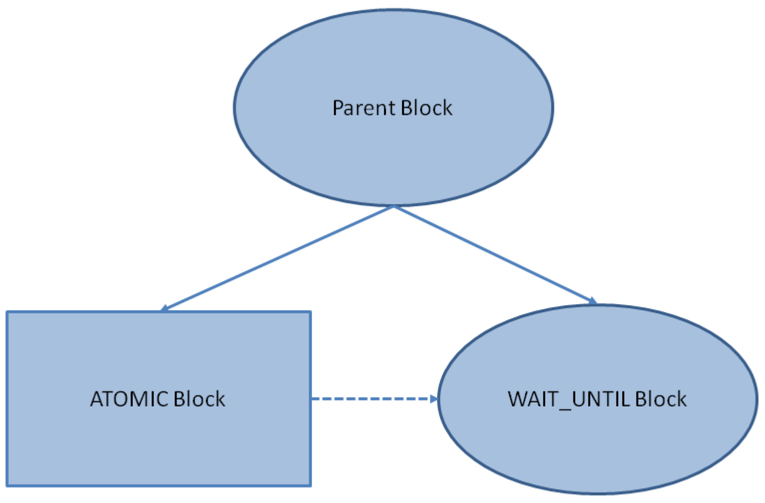

Figure 3.4: Wait Until block expression

what explained above, while Figure 3.5 shows the related state machine.

```
match parent_state with
...
| wait_until(cond) ->
  if cond then
    inc(parent_state)
    reevaluate_rule()
  else ()
| next_state ->
  //<- after running the else block the current state is this
```

3.3.3 Yield state machine

The `yield` statement block generation is similar to that of `wait`, but the state machine is quite different. Figure 3.6 shows the block scheme.

The state machine updates the fields passed as arguments of the `yield` statement by generating the AST code for field assignments, then increments the parent state and re-evaluates the rule. Figure 3.7 shows a scheme of the state machine.

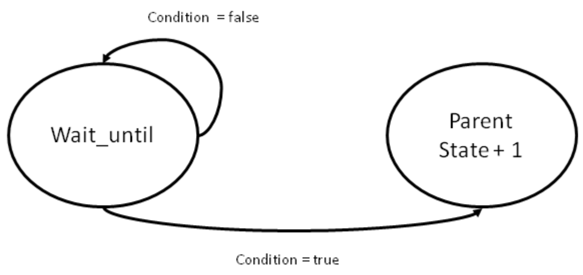

Figure 3.5: Wait Until state machine

```
match parent_state with
...
| yield v1,v2,...,vn ->
  f1 := v1
  f2 := v2
  ...
  fn := vn
  inc(parent_state)
  reevaluate_rule()
| next_state ->
  //<- this state is executed after the field update
```

3.3.4 Foreach state machine

The `for` statement block is structured along three levels: the outer level is the *for block* which has as child a *body block*. The body block has as child an *atomic block*. This is shown in Figure 3.8.

The state machine is more complex than those seen until now, as it is made of 4 states. The first state initializes a counter used to iterate the elements of the list and then it immediately re-evaluate the rule, incrementing the state of the `for` by 2. The second state executes the body and then increments the state of the for by 1. The third state checks if we have reached the end of the list. If we are at the end then it increments the state by 1 and re-evaluate the rule. Otherwise it increments the counter and set the state to 1, in order to run the body on the next element of the list. The fourth and last state exits the `for` loop. It reset the state of the loop and increase the parent state by 1, re-evaluating the rule.

```
match parent_state with
...
  | Foreach(l) ->
    for_state := 0
```

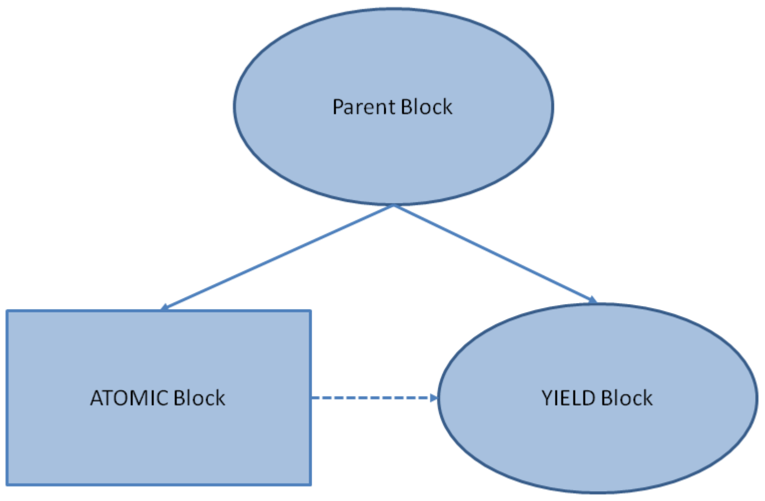

Figure 3.6: Yield block expression

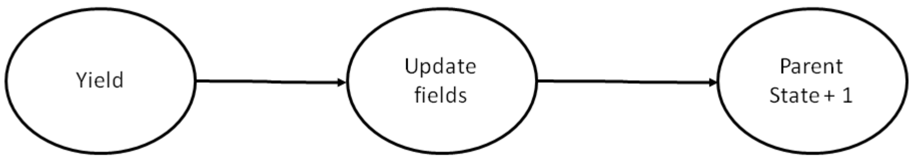

Figure 3.7: Yield state machine

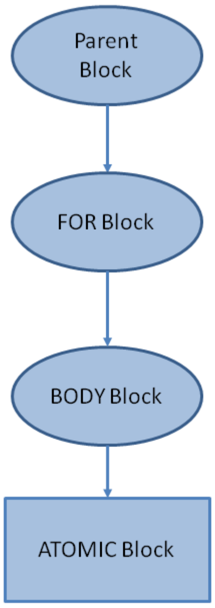

Figure 3.8: For block expression

```
    match for_state with
    | 0 ->
      ForCounter := 0
      for_state := 2
      reevaluate_rule()
    | 1 ->
      execute_body()
      inc(for_state)
    | 2 ->
      if ForCounter < l.Length then
        for_state := 1
        reevaluate_rule()
      else
        for_state := 3
        reevaluate_rule()
    |_ ->
      for_state := 0
      inc(parent_state)
  | next_state -> ...
```

Figure 3.9 shows a scheme of the state machine.

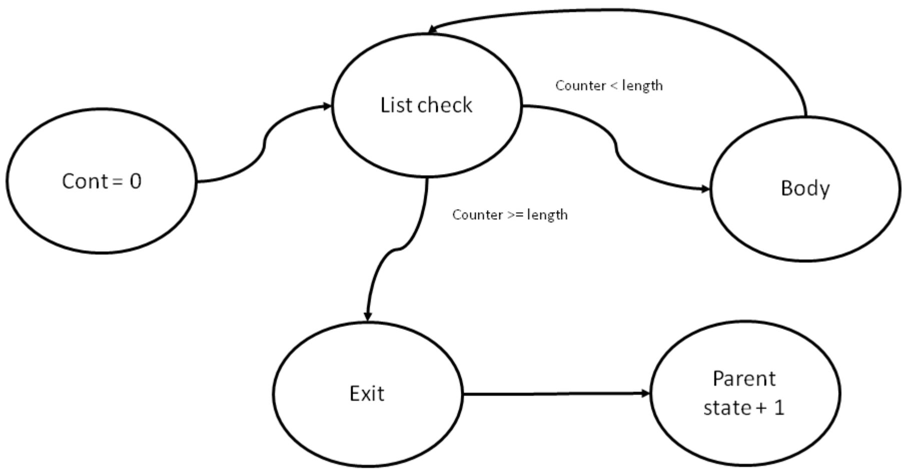

Figure 3.9: For state machine

3.3.5 While state machine

The `while` statement has 3 blocks, just like the `for` statement: the outer block is a *while block*, the second block is a *body block*, and the third block is an *atomic block*. Figure 3.10 shows a scheme of the block hierarchy.

The while state machine is made of 3 states: the first state checks the while condition. If the condition is false then we increase the while state by 1 and we re-evaluate the rule. Otherwise we increase the while state by 2 and re-evaluate the rule. The second state executes the body and then it decreases the state 1, in order to go back to the state of the condition check. The third state exits the loop, increasing the parent state. Figure 3.11 shows the state machine scheme.

```
match parent_state with
...
  | While(cond) ->
    while_state := 0
    match while_state with
    | 0 ->
      if cond then
        while_state := 2
```

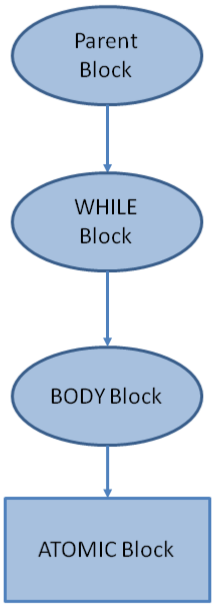

Figure 3.10: While block expression

```
        reevaluate_rule()
      else
        while_state := 1
        reevaluate_rule()
    | 1 ->
      execute_body()
      dec(while_state)
    |_ ->
      inc(parent_state)
  | next_state -> ...
```

3.3.6 If state machine

The `if` statement is made of 5 blocks: the outer block is the *if block*, whose children are *then block* and *else block*. The remaining two are *atomic blocks* whose parents are respectively the *then block* and the *else block*. This is shown in Figure 3.12.

The state machine has 5 states. The first state evaluates the condition of the `if`, If it is true then it increases the state by 1 and re-evaluates the rule. If it is false it sets the state to 4 (to jump to the `else` case) and re-evaluates

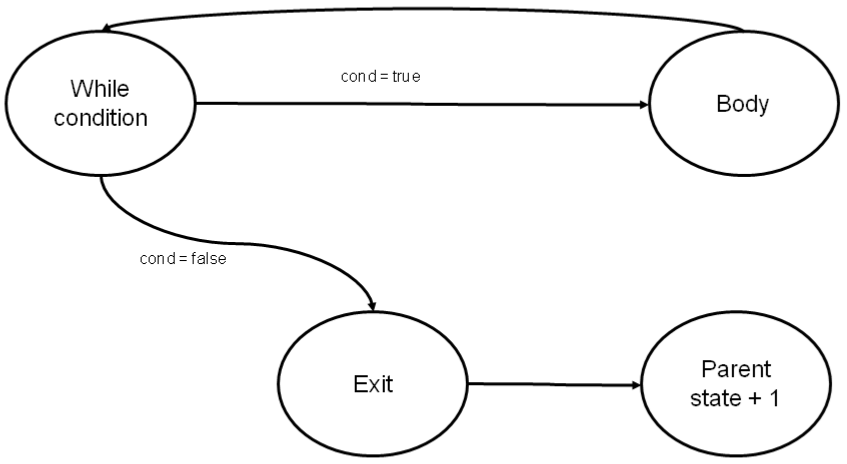

Figure 3.11: While state machine

the rule. The second state executes the `then` block, and then increases the state by 1. The third block is merely a jump to state 5, so it just sets the state to 5. The fourth state executes the `else` block and then increments the state by 1. The fifth state exits the `if` block, incrementing the parent state and re-evaluating the rule. Figure 3.13 shows a scheme of the state machine.

```
match parent_state with
...
  | If(cond) ->
    if_state := 0
    match if_state with
    | 0 ->
      if cond then
        inc(if_state)
        reevaluate_rule()
      else
        if_state := 3
        reevaluate_rule()
    | 1 ->
      execute_then_body()
      inc(if_state)
    | 2 ->
```

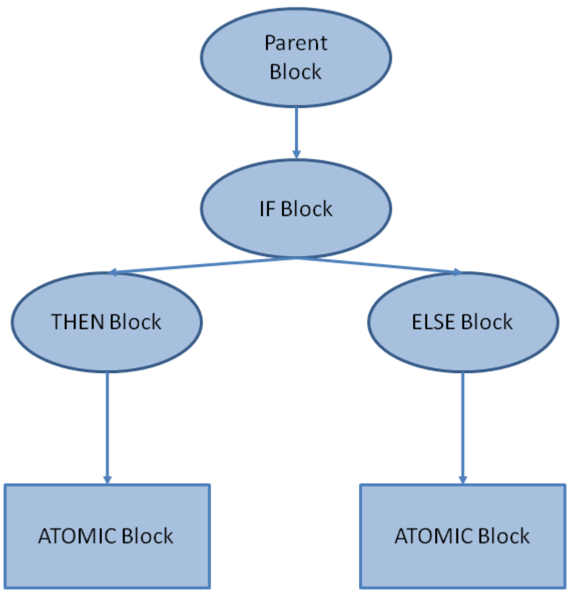

Figure 3.12: If block expression

```
        if_state := 4
        reevaluate_rule()
      | 3 ->
        execute_else_body()
        inc(if_state)
      | 4 ->
        inc(parent_state)
        reevaluate_rule()
  | next_state -> ...
```

3.3.7 Let bindings and atomic blocks

The remaining blocks have just one state and are not explained in detail, since they just executes their code and then re-evaluate immediately. However, a particular care must be done when dealing with *let bindings*. Let us consider the following code snippet:

```
rule F =
  let x = 10
  wait 1.0f<s>
```

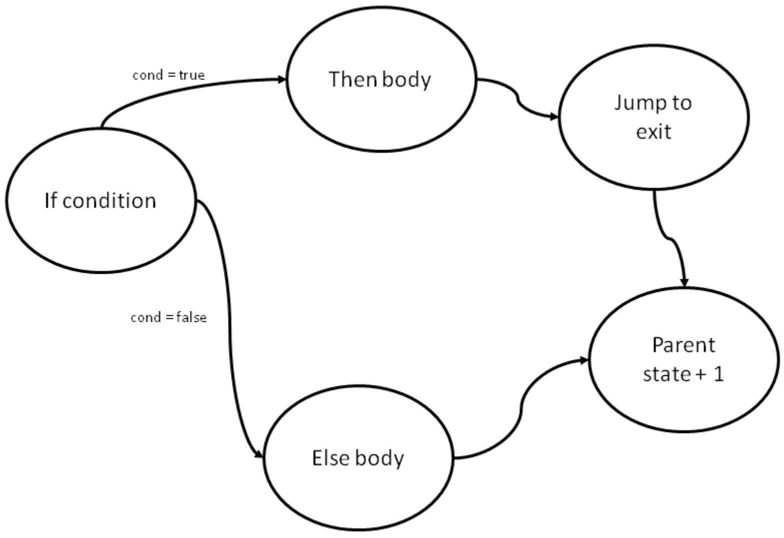

Figure 3.13: If state machine

```
  yield x
```

The yield statement cannot see the definition of `x` because the state machine definition is the following:

```
match s with
| 0 ->
  let x = 10
  if tk < 1.0f then
    tk := tk + 1
  else
    reset_timer(tk)
    s := s + 1
    reevaluate_rule()
| 1 ->
  yield x
```

so the `let` is within a different case of the `match`. For this reason every binding is mapped into a field in the entity defining the rule.

3.4 Shadowing

Mapping Casanova AST into F# AST is, in most cases, a straightforward process. All the mapping functions are not described for brevity, as they are merely functions that map a Casanova AST type into a F# AST type, filling the unused fields by Casanova with default values in F#. Particular care must be put when dealing with *shadowing* within rules. Shadowing is a property of programming language for which local variables hide within the current scope another variable with the same name in an outer scope. In Casanova we do not have the concept of variable, however we do have the concept of *let bindings*. Let bindings behave in the same way of shadowed variables, with the exception they can be shadowed within the same scope.

Let us consider now the following code

```
rule R =
  let x = 10
  yield 0
  let x = "text"
  yield 1
```

which is mapped into the following state machine, according to the `let` translation defined in Section 3.3.

```
entity E =
{
  mutable R : int
  mutable x : int
  mutable x : string
  mutable state = 0
  rule R =
    match state with
    | 0 ->
      x := 10
      R := 0
      state <- 1
    | 1 ->
      x := "text"
      R := 1
      state := 2
    | 2 ->
      state := 0
}
```

which is wrong because we cannot use the same name for two variables. We can rename the shadowed bindings progressively in the following way: we use a set of dictionary whose key is the id of the variable and the value a counter which counts the occurrences of the same shadowed variable. Whenever we encounter a `let` binding, we check if the dictionary contains that id. If it does not we add it to the dictionary and set the counter to 1. If it exists we increment the counter and store the id as `id + (string counter)`. In the example above we would get a dictionary with `x1` and `x2`.

Unfortunately this is not enough, since we still have to declare the type of the field. At this point we do not know the type of the let binding, since the type checker has not been run yet. We create an auxiliary class as follows

```
[<Nullable>]
type Ex(val) =
  let elem = val
  this.Value = elem
```

The name of the class is generating concatenating the name of the entity id with the one of the let binding id. Each variable in let bindings will have its own auxiliary class. The class is marked as *nullable* because when the entity is being constructed, we do not know the initialization value, which is set to null. The class will be instantiated at the first occurrence of the let binding. The above example will then become:

```
[<Nullable>]
type Ex1(val) =
  let elem = val
  this.Value = elem

type Ex2(val) =
  let elem = val
  this.Value = elem
entity E =
{
  mutable R : int
  mutable x1 : Ex1
  mutable x2 : Ex2
  mutable state = 0
  rule R =
    match state with
    | 0 ->
      x1 := Ex1(10)
      R := 0
      state := 1
    | 1 ->
      x2 := Ex2("text")
      R := 1
      state := 2
    | 2 ->
      state := 0
}
```

Chapter 4

Case study and evaluation

In this chapter we will present a serious game implemented in Casanova 2.0, an example showing how numerical approximation in rules can affect the behaviour of entities, and a simpler case study used for evaluation. The first is a game developed in cooperation with Tilburg university to detect dyslexia in pre-school children [6]. The last is a simple asteroid shooter where a ship shoots lasers at asteroids closing in.

4.1 Implementing a game in Casanova 2.0

This serious game uses a set of sounds to detect dyslexia in children. The sounds are played in pairs, and the user must tell if they are the same or not clicking on two different buttons on the screen. The game features a planet, consisting of two circles. The first represents a walkable landscape, while the second is a far landscape. The walkable landscape features a series of trees. A fox will close to each tree and play the first sound of the current pair. After that a bird will fly in, land on a branch and plays the second sound. After that the user will be able to click on the buttons and decide whether the sounds are the same or not. After an answer is given, the both the walkable landscape and the far landscape will rotate, to simulate the walking fox, which will play an animation. Some of the tasks are tutorial tasks, meaning that the game will auto-complete them to show the user how the game works. Figure 4.1 shows a screenshot of the game. In what follows, we will focus on the core logic of the game, skipping all the entities definition.

We begin by defining the world structure: our world is made of a series of tasks (which has type `query`), a *Graphic User Interface* (GUI) `Answer` made of two buttons, a fox, a reference to a bird, a reference to the active song, a boolean to check if the fox is playing a song, two Layers to draw

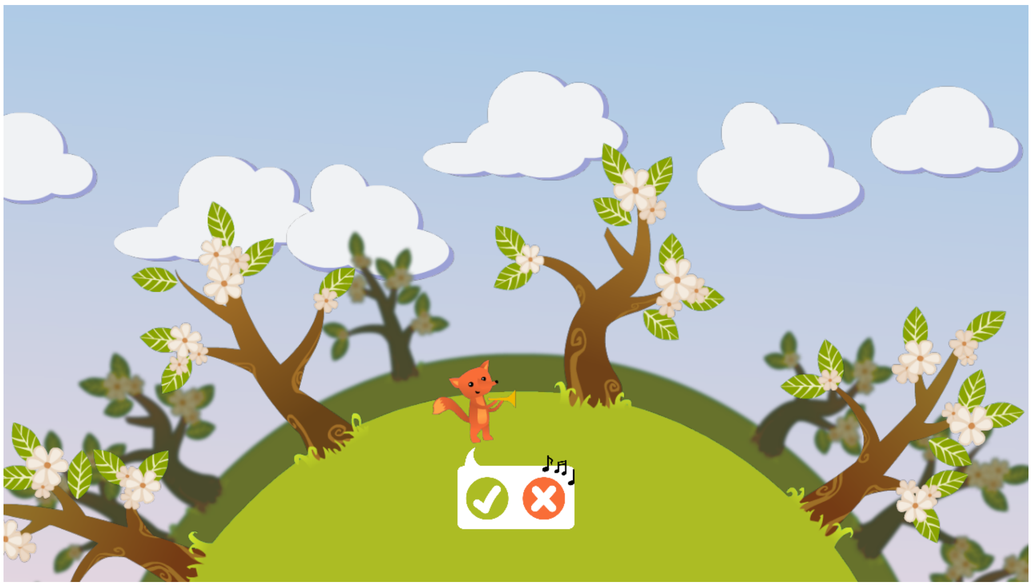

Figure 4.1: Dyslexia game in action

the landscape, a task index marking the current active task, a set of parsed tasks (read from a .csv file) and a report data structure used to write the test results in an output .csv file. We will assume that all the non-primitive types are defined elsewhere as entities or in external libraries and that they implement methods to be drawn. Also the field `Game` is used by Casanova game loop to create the game.

```
world DyslexiaDetector =
  {
    Game                        : CasanovaGame<DyslexiaDetector>
    TaskIndex                   : int
    Layers                      : ParallaxBackground
    Tasks                       : Task query
    ParsedTasks                 : List<int * string * int * bool>
    Answer                      : Button
    Fox                         : Fox
    ref Bird                    : Bird
    ref ActiveSong              : Song
    Playing                     : bool
    Report

    ...
  }
```

Now we must define a constructor for the world, which is created when the game is started. The constructor uses an external .NET library which provides a method `parseInput` to read a .csv file and build a task in a format accepted by the game. It also initializes the fox and the bird in the correct position. The bird is created with scale 0, so it is invisible at the beginning of the game.

```
Create(game : CasanovaGame<DyslexiaDetector>) =
        let date = System.DateTime.Now
        let year = string date.Year
        let month = string date.Month
        let day = string date.Day
        let hour = string date.Hour
        let minute = string date.Minute
        let second = string date.Second
        let fox_size = Vector2<pixel>.Create(150.0f)
        let file_name = day + "_" + month + "_" + year + "_" +
            hour + "_" + minute + "_" + second + ".csv"
        let tasks = InputParser.parseInput(@"Content\tracks.csv"
            )
        let fox_pos = Vector2<pixel>.Create(-50.0f,150.0f)
        let full_screen = true
        let horizontal_resolution = 1280
        let vertical_resolution = 768
        //let game = CasanovaGame.Create(full_screen,
            horizontal_resolution,vertical_resolution)
        let background = ParallaxBackground.Create(game.
            default_layer)
        let precache_y_buttons = Sprite.Create(background.
            ForegroundCanvas,Vector2<pixel>.Zero,Vector2<pixel>.
            Create(250.0f),@"yes_frames")
        let precache_n_buttons = Sprite.Create(background.
            ForegroundCanvas,Vector2<pixel>.Zero,Vector2<pixel>.
            Create(250.0f),@"no_frames")
        {
            Game = game
            Tasks = empty()
            ParsedTasks = tasks
            TaskIndex = 0
            Layers = background
            Answer = Button.Create(background.ForegroundCanvas,
                Vector2<pixel>.Create(0.0f,390.0f))
            Fox = Animal.Create("fox_walking","fox_singing", "
                fox",4,15,0.3f<s>,0.08f<s>,background.
                ForegroundCanvas,fox_pos,Vector2<1>.Create(1.5f
                ,1.5f),fox_size,fox_pos)
            Bird = Animal.Create("bird","bird","bird",1,1,1.0f<s
                >,1.0f<s>,background.ForegroundCanvas,Vector2<
                pixel>.Create(0.0f,110.0f),Vector2<1>.Zero,
                Vector2<pixel>.Zero,Vector2<pixel>.Zero)
            ActiveSong = Song.Create(game.Content)
            Playing = false
            Report = FileWriter.Create("Report",file_name)
        }
```

The game world defines a single rule which acts on most of its fields.

```
rule  Layers.RotationVelocity,
    Layers.RotationAngle,
    Answer.TutorialAnswer,
    Bird,
    Bird.Moving,
    Bird.Singing,
    Bird.Destination,
    Fox.Moving,
    Fox.Singing,
    ActiveSong,
    TaskIndex,
    Report.WriteLine,
    ActiveSong.Playing,
    Bird.SpriteScale,
    Game.Quit,
    Answer.YesPressed,
    Answer.NoPressed,
    Answer.CorrectAnswer,
    Answer.PlayTutorial = ...
```

This rule will immediately suspend itself, waiting that the list containing the task definitions is filled. This is done in order to avoid a premature exit of the game. After that we write in the report the table titles, yielding the string in the field property `WriteLine`.

```
wait_until Tasks.Length > 0
//create the column titles in the report
yield Report.WriteLine <- "Task Number,Result,Answer time"
```

We now define a `for` iterating over all the tasks

```
for t in Tasks do
  ...
```

Now the world must rotate until the tree is in the correct position. The final angle of the rotation is given by the formula

$$\alpha = -\frac{2\pi i}{N}$$

where i is the current task index and N is the number of the trees in the landscape (which we know to be 11). The negative sign is due to the fact the world turns clockwise, and in our reference system the vertical axis is inverted than a cartesian system.

At the same time the fox must start walking and we have to pre-cache the bird from the current task.

```
yield
      Bird <- t.Bird
      ActiveSong <- t.Song
      Layers.RotationVelocity <- -1.0f<rad/s>
      Fox.Moving <- true
      let target_rotation_angle = (float32 TaskIndex) * (two_pi/
          11.0f) * -1.0f
      wait_until (Layers.RotationAngle + Layers.RotationVelocity
          * dt <= target_rotation_angle)
```

The code snippet above sets the bird to `t.Bird`, that is the bird relative to the current task, the `ActiveSong` to the current task song. We then set a flag in the `Fox` entity which signals to play the walking animation. We then wait until the rotating landscape has reached the target angle.

```
yield
      Layers.RotationVelocity <- 0.0f<rad/s>
      Layers.RotationAngle <- target_rotation_angle
      Bird.Moving <- true
      Fox.Moving <- false
wait 0.1f<s>
```

When the rotation is complete, we reset the velocity to 0 to stop the rotating landscape, and set manually `RotationAngle` to the target rotation angle to correct numerical errors. We then set the fox flag to `false` to make it play her standing animation, and we set the bird moving flag to true to make it fly in.

```
wait 0.1f<s>
//wait until the bird is in position
wait_until (Bird.Velocity.Length = 0.0f<pixel/s>)
//bird is on the branch, changes its state and starts singing
yield
      Bird.Moving <- false
      Fox.Singing <- true
      ActiveSong.Playing <- true
```

The code above wait until the bird is in position, i.e. when it has stopped. Then, we set its flag to `false` and we set the singing flag of the fox and the playing flag of the active song to `true`, to play the first part of the sound.

```
wait 1.5f<s>
yield
      Bird.Singing <- true
      Fox.Singing <- false
      wait 1.5f<s>
```

```
    yield Bird.Singing <- false
```

We wait 1.5 seconds, which is the duration of each sound part, and then we set the fox singing flag to `false` to stop the singing animation, and we set the bird singing flag to `true` to start the singing animation. We wait for the sound to be played and then we stop the bird singing animation.

```
//wait for the user to answer
let start_time = System.DateTime.Now
wait_until (Answer.YesPressed || Answer.NoPressed || t.
    IsTutorial)
let user_answer =
    if Answer.YesPressed then
        1
    else
    0
let end_time = System.DateTime.Now
let elapsed_time = end_time - start_time
let elapsed_time = string elapsed_time

//bird flies away
yield
  Bird.Destination <- Vector2<pixel>.Create(-627.0f,-625.0f)
    Bird.Moving <- true
```

Now we wait until the user has pressed either one of the buttons in the interface. If the current task is a tutorial, then we will simply move ahead as the condition is automatically satisfied. The following lines measure the answer time. After that we set the bird destination and we make it fly away, by turning the moving flag to `true`.

```
let report_answer =
  if (user_answer = 1 && t.Answer = 1) || (user_answer = 0 && t.
      Answer = 0) then
      "Correct,"
    else
        "Wrong,"

//play tutorial
if t.IsTutorial then
    wait 1.0f<s>
    yield
      Answer.TutorialAnswer <- t.Answer
        Answer.PlayTutorial <- true
    wait 2.0f<s>
    yield
        Answer.PlayTutorial <- false
        TaskIndex <- TaskIndex + 1
    //update report
```

```
else
  let report_msg = (string (TaskIndex + 1)) + "," +
      report_answer + elapsed_time
    yield
      TaskIndex <- TaskIndex + 1
      Report.WriteLine <- report_msg
```

The end part of the rule produces the output in the .csv file. If the current task is a tutorial we play the tutorial now. Otherwise we increment the task index and we write the report output in the file.

When all tasks have been processed the game quits calling

```
yield Game.Quit <- true
```

at the end of the rule.

4.2 A simulator for projectile dynamics

In this section we present a simple simulator to show how Casanova 2.0 can be used to build physics simulations and how numerical approximation in rules can affect the behaviour of entities in this context. Our simulator replicates a typical phenomenon studied in general physics courses: the projectile dynamics. Using what has been introduced in Section 2.1 on the approximation of differential equations, we will run the simulation for three different projectiles, whose position is updated at every frame while their velocity is updated every $t_i > dt$. Since the approximation is less accurate we should see not only a more abrupt change of direction, but also a different trajectory.

We will start by defining the *world* of our simulation

```
world Simulator =
  {
    Game           : CasanovaGame<Simulator>
    Scene          : Canvas
    Projectiles    : Projectile query
    LogFile        : CSVMaker
    FileColumns    : int

  ...
  }
```

The field `Game` has type `CasanovaGame<'w>`, where 'w is the type of the world entity. It is passed to the game engine to update the rules. We must define `Scene` which has type `Canvas` to draw our entities. A *Canvas* is a built-in component which contains the graphics data for the object drawn on the screen. The display is made of layers which allow to simulate the depth of view in a 2D environment. A Canvas is used to define which objects belong to the same layer.

We then define a list of `Projectiles`, whose type is a query of projectiles. The query is a special kind of list which enables the usage of a SQL syntax to operate on its element, which we will see below. The `Projectile` type is an entity type defined below.

The other two fields are used to create a CSV file containing the data on the projectiles, which will not be covered for brevity.

We now give the definition of the projectile entity:

```
entity Projectile =
  {
    Position          : Vector2<pixel>
    Velocity          : Vector2<pixel/s>
    Acceleration      : Vector2<pixel/s^2>
    UpdateRate        : float32
    Tracers           : Dot query
    TracerColor       : Casanova.Drawing.Color
    Sprite            : Sprite
    ref Scene         : Canvas
    Log               : string
    LineCounter       : int
    Id                : int

    ...
  }
```

The projectile has a Position, a Velocity, and an Acceleration, whose type is $Vector2 < measure >$ which is a built-in type. Units of measure are a feature of Casanova. Since the screen coordinates are measured in pixels, the position measure is a pixel, the velocity is $pixel/s$ and the Acceleration is $pixel/s^2$. The Update Rate is the rate at which the velocity of the projectile will be updated. The tracers are a series of static entities, which represents the dots drawn on the screen to mark the trajectory. We will not show their definition in detail for brevity, since they are not updated by the game engine, having no rules. The Tracer Color is the colour of the dots. A Sprite is a built in type containing the information on the size and the position of the image representing the object and the image file associated with it. The other are used to generate the report file. We now have to define a constructor for the projectile. We want be able to dynamically define its starting position, velocity, acceleration (which will be only along the Y-axis to simulate gravity), the rate at which the velocity is updated, its Canvas, the color of the trace, and an id to uniquely identify it[1].

```
Create(position ,velocity ,acceleration ,update_rate ,scene
    : Canvas,tracer_color ,id ,technique) =
      {
        Position = position
        Velocity = velocity
        Acceleration = acceleration
        UpdateRate   = update_rate
        Tracers = empty()
        TracerColor = tracer_color
        Sprite = Sprite.Create(scene ,Vector2<pixel >.
           Create(0.0f ,0.0 f) ,Vector2<pixel >.Create(48.0
           f ,12.0f) , "projectile.png")
        Scene = scene
        Log = ""
        LineCounter = 0
        Id = id
        Technique = technique
      }
```

Now let use define a rule to update the position according to the Euler scheme defined in Section 2.1.

```
rule Position = yield Position + Velocity * dt
```

The position is updated at every frame, since we want the representation of the trajectory to be as accurate as possible.

As for the Velocity, we want to update it at different rates for all projectiles. Our approximation interval will then be $\Delta t = dt \cdot UpdateRate$. The rule definition is the following:

```
rule Velocity =
  yield this.Velocity + Acceleration  * UpdateRate * dt
  wait UpdateRate * dt
```

We update the velocity according to the differential equation defined in Equation 2.1. We, then wait for $\Delta t = UpdateRate \cdot dt$ seconds to stop the simulation.

[1]Again, the `Log` is used to generate the report and will not be explained in detail

The following rules draws the projectile and its trajectory on the screen

```
  rule Tracers =
    let tracer = Dot.Create(this.Position , this.Scene ,
       TracerColor)
    yield tracer + Tracers
    wait 0.1f<s>

  rule Sprite.Position = yield Position

  rule Sprite.Rotation =
    if Velocity.X = 0.0f<pixel/s> && Velocity.Y > 0.0
      f<pixel/s> then
      yield 3.0f * pi_over_2
    elif Velocity.X = 0.0f<pixel/s> && Velocity.Y <
      0.0f<pixel/s> then
      yield pi_over_2 * 1.0f
    else
      yield atan(Velocity.Y / Velocity.X) * 1.0f<rad>
```

The first rule creates a dot every $0.1s$, the second one clamps the image of the projectile to its position to update its representation on the screen. The second rotates the projectile image according to the velocity vector. We use the well known formula to find the angle of the vector with respect to the X-axis, which is

$$\Theta = \arctan\left(\frac{v_y}{v_x}\right) \tag{4.1}$$

Since Equation 4.1 fails for $v_x = 0$, we consider those cases separately and immediately return the angle associated to $v = [0, v_y]$.

Now that we have defined the dynamics of the projectile, we need to update the world. We first need to create our projectile.

```
  rule Projectiles =
    wait_until Projectiles.isEmpty
    let test_shell1 = Projectile.Create(Vector2<pixel
      >.Create(−400.0f,250.0f),Vector2<pixel/s>.
      Create(100.0f,−300.0f),Vector2<pixel/s^2>.
      Create(0.0f,98.1f),1.0f, this.Scene, Color.Blue,
      Projectiles.Length)
    yield test_shell1 + Projectiles
    let test_shell2 = Projectile.Create(Vector2<pixel
       >.Create(-400.0f,250.0f),Vector2<pixel/s>.
       Create(100.0f,-300.0f),Vector2<pixel/s^2>.
       Create(0.0f,98.1f),30.0f,this.Scene,Color.Red,
       Projectiles.Length)
    yield test_shell2 + Projectiles
```

The following rules waits until the projectile list is empty. This is needed in order to loop the simulation and to re-create the projectiles when they are deleted because they lies outside the screen boundaries (see below). It creates two projectiles with initial position $p = [-400, 250]$[2], initial velocity $v = [100, -300]$, and a constant acceleration $a = [0, 98.1]$ (gravity). The first projectile velocity is updated 60 times per second, the other one only two time per second.

When the projectiles go outside the screen boundaries, we remove them. After that the previous rule generates two new projectiles. This is achieved by the following rule:

```
  rule Projectiles =
    wait_until not Projectiles.isEmpty
    let screen_boundary = 800.0f<pixel>
    let out_of_screen_projectiles =
      from projectile <- Projectiles
        where projectile.Position.X > screen_boundary
            || projectile.Position.X < (-
          screen_boundary) ||
              projectile.Position.Y > screen_boundary
                || projectile.Position.Y < (-
              screen_boundary)
        select projectile
    if Projectiles.Length = out_of_screen_projectiles
       .Length then
      yield empty()
```

Finally, the following rule stops the simulator and generates the report file when pressing escape.

```
  rule Game.Quit,LogFile =
    wait_until is_key_down Keys.Escape
    for p in Projectiles do
```

[2]Note that in the screen coordinate system the Y-axis is inverted as opposed to the cartesian system.

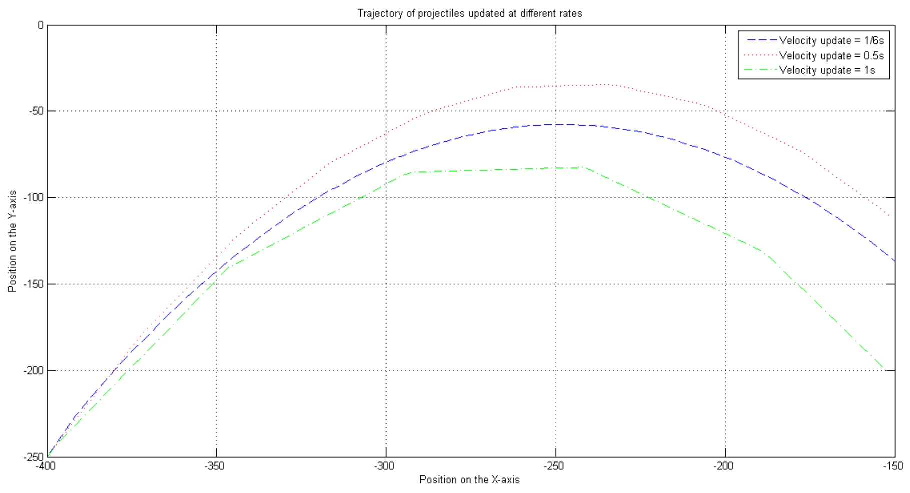

Figure 4.2: Chart of the trajectory at different update rates

```
      yield LogFile.WriteLine <- p.Log
    wait_until not LogFile.Writing
    yield Game.Quit <- true
```

4.2.1 Experimental results

We ran the simulation using 3 projectiles with an update rate of $1/6s$, $1/2s$, and $1s$. We can appreciate the fact that the trajectory drastically changes due to a worse approximation of the velocity, and that the transition when updating the velocity is less smooth than at higher rates. Figure 4.2 shows the trajectory of projectiles at the different update rates. Figure 4.3 shows the simulator in action. We note that the trajectory is different with respect to the update rate, and that, at lower update rates, the shape of the position function resembles more a piecewise linear polynomial than a parable[3], as it should be for the trajectory of a projectile. This is due to the fact that the position is updated at every frame, but not the velocity. Thus, when the velocity is not updated, the projectile travels at constant speed (linear motion). When the velocity is updated, the trajectory changes according to the deceleration, abruptly in the case of a lower precision (lower update rate).

[3]Even though, at a higher precision, we still obtain a piecewise linear polynomial and not the exact function

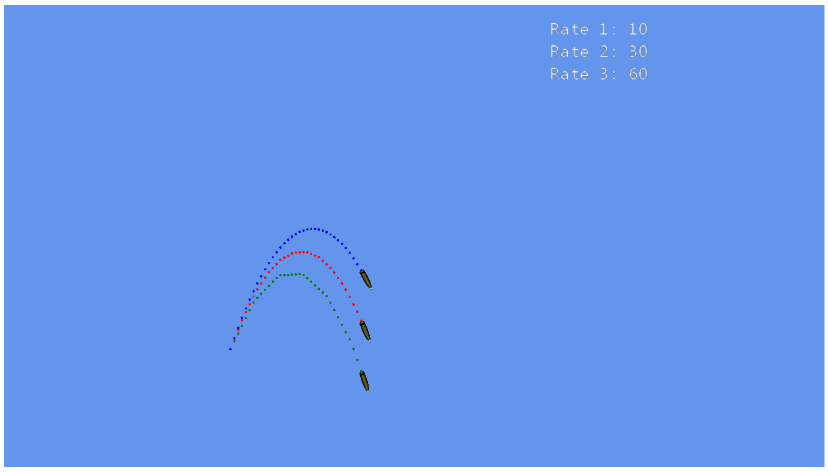

Figure 4.3: Running simulator showing the trajectories of the projectiles at different update rates

4.3 Case study and evaluation

We will evaluate Casanova 2.0 comparing the new implementation (compiled) of the *asteroid shooter* game with the old one (interpreted) written in Casanova 1.0. Since the highest computational time is spent by computing the collisions among asteroids and lasers, we will measure the processed frames per second in function of the number of asteroids. The benchmark version of the game will shoot a laser at each frame, create a certain number of consecutive asteroids, and finally pause the generation process for 1 second. Table 4.1 shows our results.

We note that Casanova 1.0, although performing quite the same for few entities, has a huge performance drop when dealing with a large amount of collisions, at the point the game it becomes almost unplayable (to have a smooth game play experience the fps rate should be higher than 30) even for a moderate amount of entities. On the other hand, Casanova 2.0 does not suffer relevant performance drops even when dealing with huge amounts of entities in the game state. This result is strengthened by the fact that the asteroid shooter implemented in Casanova 1.0 makes use of index optimization on the queries to detect collisions, which have not been implemented yet in Casanova 2.0.

This result clearly shows the loss of performance using scripts implemented with monads compared to our compiled implementation of a state machine for interruptible rules. Figure 4.4 shows a chart of the average processed FPS in both versions of the language.

Casanova 1.0			
asteroids	min fps	max fps	avg fps
100	40	60	60
250	15	60	20
500	10	50	11
1000	5	50	7
5000	2	48	5
Casanova 2.0			
100	51	59	57
250	50	59	57
500	50	59	49
1000	39	59	47
5000	37	59	40

Table 4.1: Benchmarks on the asteroid shooter

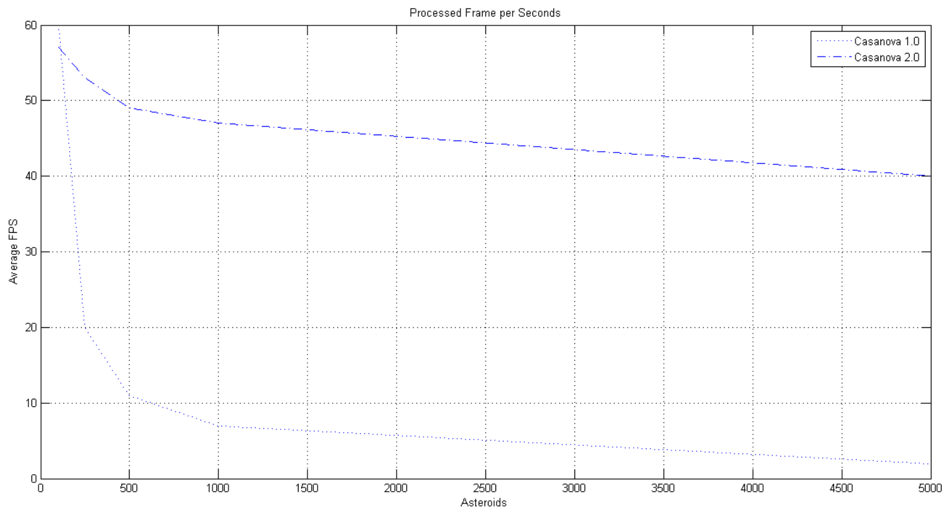

Figure 4.4: Chart of average FPS processed by both asteroid shooter implementations

Chapter 5

Conclusions and future works

In this thesis work we presented Casanova 1.0, which was interpreted and written in F#, and exposed its flaws. We analysed the problem of having implemented a scripting language using a monadic domain specific language, where each lambda abstraction is translated into a class by the F# compiler. Besides our surveys showed that testers tended to "abuse" the rule structure even to implement timers, using scripts just to define input events. This has brought us to redefine our language, merging the concepts of rules and scripts into a single structure called *interruptible rule*. Besides we built a compiler, based on the F# one, to solve the problem related to monadic scripts, implementing a state machine for interruptible rules. The first non-trivial challenge we faced was to understand the basics of the F# lexer and parser to extend the language introducing Casanova keywords and syntax definitions. The main problem with the lexer/parser is that F# is a context-dependent language, since it uses indentation as a syntax rule to define blocks (and not delimiters such as `begin/end` or curly brackets), so the lexer must use a stack of auxiliary tokens to detect the block boundaries. The second challenge was to design the state machine for rules. We started from a trivial implementation, where each statement was a state in the automaton, and then implemented an improved version where only time primitives or state updates interrupt the rule. The third challenge was solving the problem of scoping in the state machine. Indeed, the state machine introduces additional scopes and blocks that are not defined in the source code of the program. We had to create an automated system to map the local variables into a global scope visible within the rule. A further challenge was to solve the problem related to shadowing, where multiple definitions of the same variable might occur during the translation of the rule to the state machine. Finally we presented a serious game implemented for a team of researchers of Tilburg university to detect dyslexia in pre-school children. We used a benchmark sample to test the performances of Casanova 2.0 compared with the same implementation in Casanova 1.0. The results showed that Casanova 2.0 can process an amount of frames per second which is even 6 times larger than Casanova 1.0 with a high number of entities in the game.

Future works on this subject will be the following:

- Internet multi-player aspect of games. Indeed the network synchronization of the game state is a hard and complicated subject, and no high-level libraries, which provide the programmer with an interface to send data, exist. We are planning to extend Casanova language with network primitives which implements a system of message passing between rules run on different machines. A first attempt had already been made in Casanova 1.0, although never completed, which used a modified version of the world traversal presented in Section 2.4.4 to decompose the game world in atomic values which could be sent by a network library written in .NET.

- Code optimization through static analysis techniques. We will employ static analysis to analyse the code and determine correctness and performance properties which the programmer may use to improve its code. Particular care will be posed to query optimization techniques, i.e. identifying common patterns for optimizable queries and build indices to speed up the research, and network bandwidth usage and prediction techniques.

- Object-Oriented paradigm implementation. The current version of casanova compiler does not support the object-oriented syntax described in Section 2.7 and they will be implemented in future versions.

- Translation of Casanova into C# code, to achieve better performances, since the source code of the back-end of the compiler has been released.

- Integration of Casanova in Microsoft Visual Studio. This requires building an extension to manage the project system in Casanova, and trying to extend the intellisense to support Casanova.

Bibliography

[1] Entertainment software association. http://www.theesa.com.

[2] Game engines. http://en.wikipedia.org/wiki/Game_engine, 2014.

[3] Nwscript. http://en.wikipedia.org/wiki/NWScript, 2014.

[4] Mohamed Abbadi, Francesco Di Giacomo, Renzo Orsini, Aske Plaat, and Pieter Spronck. Resource entity action: A generalized design pattern for rts games. *The 8th international conference on computers and games (CG2013)*, 2013.

[5] R. W. Crandall and J. G. Sidak. Video Games: Serious Business for America's Economy.

[6] G. Maggiore, M. Abbadi, M. Postma, F. Di Giacomo. A serious game for detecting dyslexia in children. http://casanova.codeplex.com/.

[7] Jason Gregory. *Game engine architecture.* Taylor & Francis Ltd., 1 edition, 2009.

[8] Paul Hudak. *The Haskell school of expression : learning functional programming through multimedia.* Cambridge University Press, New York, 2000.

[9] Peter J. Kovach. *The Awesome Power of Direct3D/DirectX - DirectX 5 Version.* Manning Publications Co., Greenwich, CT, USA, 1998.

[10] Z.N. Li and M.S. Drew. *Fundamentals Of Multimedia*, chapter 10, pages 288 – 290. Prentice-Hall Of India Pvt. Limited, 2005.

[11] A. Cortesi P. Spronck D. Dini G. Maggiore M. Abbadi, F. Di Giacomo. Orchestration in games. Technical report, Università Ca' Foscari DAIS, Tilburg University, NHTV University of Breda, 2014.

[12] G. Maggiore. *Casanova: a language for game development.* PhD thesis, Università Ca' Foscari di Venezia, Dipartimento di Informatica, 2012.

[13] G. Maggiore. A proposal for a casanova 2.0 networking. Technical report, NHTV Breda University of Applied Sciences, 2014.

[14] Giuseppe Maggiore, Alvise Spanò, Renzo Orsini, Michele Bugliesi, Mohamed Abbadi, and Enrico Steffinlongo. A formal specification for casanova, a language for computer games. In *Proceedings of the 4th ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, EICS '12, pages 287–292, New York, NY, USA, 2012. ACM.

[15] Alfio Quarteroni, Fausto Saleri, and Paola Gervasio. *Scientific Computing with MATLAB and Octave, 3rd ed.* Texts in Computational Science and Engineering. springer, 2010.

[16] John Rice. Assessing Higher Order Thinking in Video Games. *Journal of Technology and Teacher Education*, 15(1):87–100, January 2007.

[17] Jason Della Rocca, Hank Howie, Steve Meretzky, Joe Minton, Kent Quirk, and Tracy Rosenthal-Newsom. In the trenches: game developers and the quest for innovation. In *Proceedings of the 2006 ACM SIGGRAPH symposium on Videogames*, Sandbox '06, pages 9–11, New York, NY, USA, 2006. ACM.

[18] Joachim Schmid. Compiling abstract state machines. *Journal of Universal Computer Science*, 7:2001.