# Game Programming as a Non-Threatening Introduction to Functional Languages

Giuseppe Maggiore
Ca' Foscari University of Venice
Computer Science Department
giuseppemag@gmail.com

Giulia Costantini
Ca' Foscari University of Venice
Computer Science Department
malvoria@gmail.com

Agostino Cortesi
Ca' Foscari University of Venice
Computer Science Department
cortesi@unive.it

## ABSTRACT

In this paper, we explain a new analysis technique to guide CS students during the development of a videogame. This technique is so structured as to make students learn the basics of functional programming without even realizing it. We start by describing some of the difficulties associated with forcing a more abstract style of reasoning and a functional language to undergraduate students. Then, we describe our general abstract framework for developing videogames and we show how an abstract model for a videogame built with this technique can be translated into working code almost effortlessly. Finally, we discuss how this approach can be extended to handle issues like users' interaction and more sophisticated state management.

## Categories and Subject Descriptors

K.3.0 [**Computers and Education**]: General, K.8 [**Personal Computing**]: Games

## General Terms

Algorithms, Design, Experimentation, Human Factors, Languages

## Keywords

Functional programming, games.

## 1. INTRODUCTION

In this paper, we present an approach based on the formal analysis of the inner logic of video-games. This approach is aimed at introducing CS students to basic concepts about functional programming and how to do abstract reasoning about code.

Why is it difficult for students to get into functional programming? First of all, learning a programming language is never easy, especially if it is based on a different paradigm from the one students have seen in their introductory language. As a matter of fact, in a traditional CS curriculum all students after the first year have already been exposed to imperative programming, often sprinkled with a bit of object orientation and they have seen this coding style work in various circumstances. Thus, many students feel that learning another paradigm and new languages is a bit of a waste of time, since they think they already know enough to solve real world problems. Moreover, students are used to write code for simple applications without really thinking in advance about the structure of the program and the domain of the problem. Unfortunately, while this approach leads to almost immediate results (for example, in a videogame that would be something moving on the screen) it also leads to messy code that can easily hide bugs of various natures. In functional programming, the stress on "thinking before writing" is as big as it can get: unstructured code rarely even compiles because of the strictness of most modern compilers. Our approach aims at showing students that functional programming can be a useful and powerful tool to achieve working, elegant and well structured programs. Even better, with a careful choice of programming tools and development environments, students are not suggested a completely alternative way of programming, but rather a new way that gracefully interoperates with the one they already knew. Finally, we believe we managed to avoid the students feeling that FP is a "*solution in search of a problem*". The building blocks of our technique are functional operators (that emerge as the most natural way to describe certain programs) but the student will not have to find this out at all.

As an application field for our technique we choose video games because they are both engaging and entertaining material for students, while also presenting some serious design and coding challenges. We tested the use of this technique on a group of third-year CS students who were preparing their bachelor degree report. The results have been very promising: the students were able to create three small games each (during a single month) and they didn't encounter any difficulty using our technique. As an added bonus, students loved the idea of making some use of mathematical concepts (from sets to induction) in a real-life setting to produce something both interesting and aesthetically appealing. This made them feel that the most theoretical parts of their study curriculum are more useful than it might seem at a first glance, and this has been perceived as quite empowering.

The structure of the paper is as follows. In the section "*Part I – Game analysis' method*" we will show how the first part of the technique works, listing its main steps and explaining their meaning. After that, we will introduce a case study ("*Case study – Puzzle Bobble*"), which can help us better understand how to apply the technique to a real game. In "*Part II – From the analysis to the code*" we will see the second part of the technique: how you can go from the mathematical and abstract analysis to source code which really works. This is a very important part of our technique, because it shows how simple it is going straight from the analysis to the working code. Once you have a well structured analysis, writing the code is almost trivial: the main focus of our technique is about thinking instead of directly jumping to code-writing. In the last section, "*Conclusions and future work*", we will share the results of our technique: how did the students react to it and what they were able to produce with this approach in a really small amount of time. We will also propose some extensions to our technique.

## 2. PART I – GAME ANALYSIS METHOD

The first part of our technique, which we can call the "pen & paper part", is strictly about logic and mathematics. The basic idea of this part is: think about the logic of your game, identify the entities involved and how they interact with each other over time.

In fact, a video game consists of a series of entities on the screen which appear, move and disappear (not necessarily in this order). Moreover, a game is made of a collection of game screens, one after the other: time is not continuous, but discrete. If we split time into steps (*frames*, to be precise), we can think of the beginning of the game (the first game screen rendered) as frame 0; the following one will be frame 1 (frames occur at between 18 and 60 Hz for a human player to have a feeling of fluid action); the next will be frame 2 and so on. We see now how we can apply the basis of the induction principle in our case: if you define frame 0 and how to go from frame `t` to frame `t + 1` you have automatically defined the entire collection of time frames.

The main steps of the analysis are thus the following:

- *Entities identification*, that is understanding which entities are relevant for the logic of the game
- *Base case*, that is the state of those entities at the beginning of the game
- *Inductive step*, that is how the entities change from a time step to the following one

### 2.1 Entities Identification

This is the very first step of the analysis: selecting the game elements. It is an important step because, on one hand, we don't want to include unnecessary elements in our analysis. On the other hand, we must be careful about including all the elements that will be necessary.

First of all, we will have to imagine the game and define a list of all the entities which will, at some point or other, appear in the game. Then, we have to discard all those entities which are not strictly necessary for game logic. Finally, we have to identify the main characteristics of each entity: is it a set, a list or a single item? Is it a tuple? Is it a scalar? And so on.

The grammar for our entities is the following:

$$Entity ::= Set\ of\ Entity \mid List\ of\ Entity \mid Value$$
$$Value ::= float \mid int \mid bool \mid \underbrace{Value \times \ldots \times Value}_{n\ \text{times}}$$

This first step of our technique can be thus summarized in the following picture:

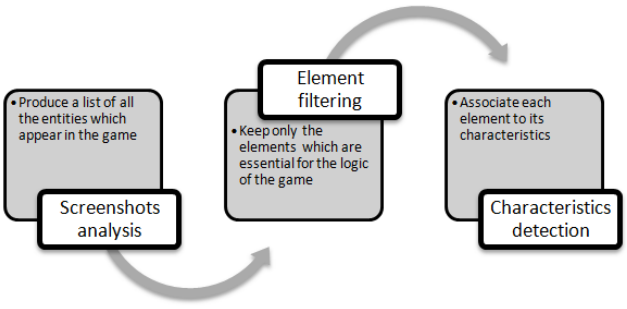


**Figure 1 - Entities Identification's Steps**

### 2.2 Base Case

In the previous step we identified all the essential entities of our game: now we have to decide the initial values of these entities. Some sets of entities might be empty, some entities will be zeroed and other entities will come from a certain description of the current game level, and so on. This part is very simple to describe in games where the initial status of the entities must not be computed procedurally. Otherwise, it will use the same language as the next step.

### 2.3 Inductive Step

As we stated before, a game consists of a succession of game screens: in this step we have to understand how our entities change from one game screen to the next one. This means that we must describe a function `play` that takes as input our entities at time `t` ($e^t_1$, $e^t_2$, …, $e^t_n$) and that returns one of the following:

- `WIN` → when the player has won the game
- `LOSE` → when the player has lost the game
- `play` ($e^{t+1}_1$, $e^{t+1}_2$, …, $e^{t+1}_n$) → when the game must continue

Inside the body of the `play` function we can define new functions, temporary entities, and so on. In particular we write functions case-wise (we will shortly see what this means) and assign entities using an intensional syntax (set comprehensions).

As an example of our syntax, consider a (not so trivial) function that takes a starting point `a` and a predicate `p` and returns the set of vertices of graph `G` that can be reached from `a`, passing only through nodes that pass the predicate `p`:

```
reach a p = reach* {a} (neighbours p {a})
  where
    neighbours p s =
      {x | a ∈ s, x ∈ (adjacent G a) ∧ p x}
    reach* s ∅ = s
    reach* s t =
      reach* (s + t) ((neighbours p t) – s - t)
```

The example above shows the definition of `reach` by cases over its input parameters (it returns `s` iff the last parameter is the empty set) and how to use the well known set-comprehension syntax to define a set of elements from sets that are already known:

```
{x | a ∈ s, x ∈ (adjacent G a), p x}
```

which is the set of vertices `x` that respect predicate `p` and which are adjacent to some vertex `a` from set `s`.

## 3. CASE STUDY – PUZZLE BOBBLE

As a case study we use Puzzle Bobble, a well-known 2D arcade game which offers the right complexity trade-off for our analysis: it is not too easy, the game mechanics are intuitive enough and imperative implementations are amazingly complex when compared with the game itself. The rules of the game are quite simple: the player can shoot colored balls towards a group of balls hanging from the ceiling. When three (or more) balls of the same color are in contact, they explode and fall off towards the ground. If, after an explosion, some balls hover, they fall off too. The objective of the game is to destroy all the balls in the screen.

## 3.1 Entities Identification

### *3.1.1 Screenshots Analysis*

To identify the entities of the game we can take a look at some screenshots (if an implementation of the game is already available) or draw some sketches of how we imagine our game should be. In this case we use two screenshots:

**Figure 2 - Puzzle Bobble screenshots**

The elements present in these screenshots are the following:

- Writings (Round, Credits, Score)
- Background
- Dinosaurs
- Bag
- Cannon
- Next ball to be shot (yellow in the left screenshot, green in the other)
- Ball in movement (blue, left screenshot)
- A group of balls hanging from the ceiling
- Balls that are falling (two blue balls in the right screenshot)

### *3.1.2 Element Filtering*

We notice immediately how some elements are not relevant to the logic of the game: writings, background, dinosaurs and bag are all related to the graphics part of the game and nothing else (they don't really influence the logic development of the game). The cannon is not relevant too, since it is mainly related to the input part of the game. In fact, the cannon rotates and shoots a ball only in relation to the user's input. The ball to be shot next stays still in a fixed position until some input is received: after that it turns into moving ball (which is a completely different entity). So this ball mainly regards graphics and input: we can exclude it too. The three remaining entities (climbing balls, hanging balls, dropping balls) are all relevant: they compose the basic logic of the game. At each frame we will have to update the status of these entities: climbing balls will move up and could collide with hanging balls, hanging balls could explode and dropping balls will move down.

To summarize, the entities we have identified are:

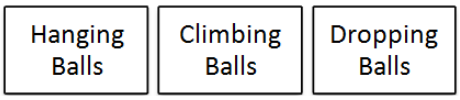


**Figure 3 - Puzzle Bobble entities**

### *3.1.3 Characteristics Detection*

Now that the relevant entities have been identified, we can move on and define their characteristics. Note that each main entity is, actually, a `Set` of entities: climbing balls is a set of balls that are moving up, hanging balls is a set of balls that are still, etc. So we will move on to defining the characteristics of these elements: balls.

Each ball (whether it is climbing, hanging or dropping) can be described with a position, a radius and a color. In addition, climbing and dropping balls also have a speed.

To summarize:

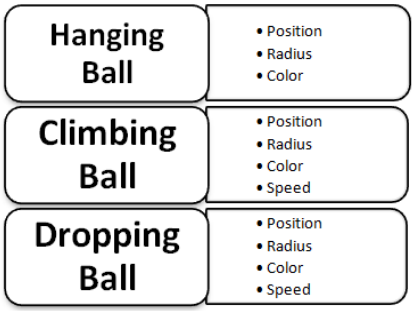


**Figure 4 - Entities' characteristics**

## 3.2 Base Case

Once the game entities have been identified, the initial values of each entity are set as follows:

- *Hanging balls*: a set of a certain number of balls, positioned in a grid on the top of the screen, with random colors;
- *Climbing balls*: empty set (when the game starts we haven't shot balls yet);
- *Dropping balls*: empty set (when the game starts all the balls are still).

$$hanging_0 : Set\ (Position \times Color \times Radius) = \dots\ (random)$$
$$climbing_0 : Set\ (Position \times Color \times Radius \times UpwardSpeed) = \emptyset$$
$$dropping_0 : Set\ (Position \times Color \times Radius \times DownwardSpeed) = \emptyset$$

## 3.3 Inductive Step

This is probably the most difficult (and important) step of the entire technique. We have to carefully analyze the logic of our game and formalize it. In particular, we have to understand how the entities modify themselves from time step `t` to time step `t + 1`.

Before starting with the analysis, remember that our game is characterized by three entities: the collections `hanging`, `climbing` and `dropping`. During the analysis we will make use of temporary entities (`stuck`, for example), but at the end of it we will remain only with `hanging`, `climbing` and `dropping`.

The update function checks whether the user has blown all the hanging balls (`WIN`), whether there is at least one hanging ball that has reached the bottom of the screen (`LOSE`) and, otherwise, updates the entities and calls itself with the updated entities as its parameters:

```
play ∅ climbing dropping = WIN
play hanging climbing dropping =
  if (∃q ∈ hanging | pos(q) = Bottom) then
    LOSE
  else
    play hanging' climbing' dropping'
    where
      hanging' = …
      climbing' = …
      dropping' = …
```

We must thus find the values of `hanging'`, `climbing'`, and `dropping'`. First, we have to consider climbing balls which collide with some other hanging balls:

```
stuck = {m | m ∈ climbing ∧
             (∃s ∈ hanging | touch m s)}
```

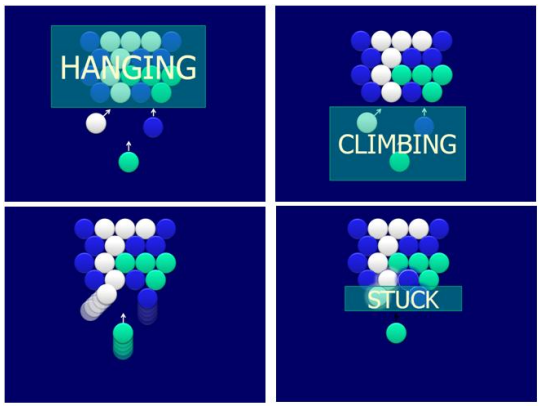


**Figure 5 – Hanging, Climbing and Stuck Balls**

We will call `stuck` the balls which must stop their upward run because they have hit a hanging ball. The climbing balls which will still be climbing during the next iteration will be those which have not become stuck and they must be updated according to their movement direction:

```
climbing' = {moveup(m) | m ∈ (climbing - stuck)}
```

Notice that we do not need to specify exactly how the `moveup` function or the `touch` function work: this is a minor issue that will be dealt with at the end of the coding process.

At this point we can expand all the `stuck` balls by color, using the `reach` procedure defined in the previous section:

```
touched = {reach s (sameColor s) | s ∈ stuck}
```

The rest of the code can be obtained similarly, getting to:

```
drop = union {s | s ∈ touched, size s ≥ 3}
where
  touched = {reach s (sameColor s) | s ∈ stuck}

hanging' = {reach s p | s ∈ may_hang ∧
                        position s = top}
where
  p x = true
  may_hang = hanging + stuck – dropping'

detached = hanging – hanging'
dropping' = dropping + drop + detached
dropping' = {fall(d) | d ∈ dropping'}
```

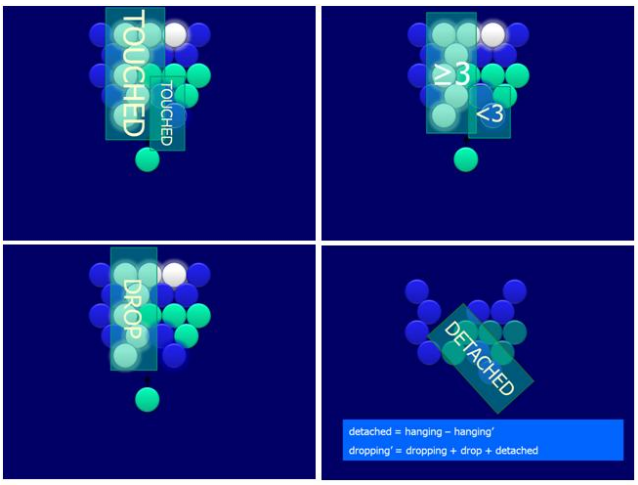


**Figure 6 – Touched, Dropping and Detached Balls**

Although the analysis we just completed was not so easy to make, we have expressed our game logic in a very powerful formalism. Moreover, each statement is very clear to read (just read it out loud and it will explain itself) and it can be unexpectedly immediate to transform this formalism into working code, together with a meaningful class hierarchy.

## 4. PART II – FROM THE ANALYSIS TO THE CODE

The first part of our proposed technique led us to a formal description of the game entities and their interactions. Now we have to translate these descriptions into source code: we will see how each step we took in the first part was preparatory for a step of this second part. In particular, the game entities and their characteristics will become classes of object; the predicates we identified (i.e., `moveup`) will become methods of those classes; the base case will be the initialization of our game, while the inductive step will form its update.

As programming language we considered two options: *F#*, a functional language with the possibility of object orientation, and *C#*, an object oriented language equipped with a very useful library (*LINQ*) of functional operators. We adopted these two languages because they are both part of the *.NET* framework, which provides useful libraries (in the case of game development we made heavy use of *XNA*) supporting development in various areas.

We will show examples of translations (from expressions to code queries) with both languages.

## 4.1 Entities to Classes

If we correctly identified entities and their characteristics, each entity will become a class, and the fields of each class will derive from the characteristics of the associated entity.

In our example, we identified three base entities: hanging balls, climbing balls and dropping balls. Thus we will have three classes (`Ball`, `ClimbingBall`, `DroppingBall`), one for each entity. The fields of `Ball` will be position, radius and color; the fields of `ClimbingBall` will be position, radius, color and upward speed; the fields of `DroppingBall` will be position, radius, color and downward speed. We immediately notice repetitions: some fields are identical in all three classes. This happens because all these objects, deep down, are simply balls. We can take advantage of this observation and better structure our code, imposing inheritance relationships between classes (for example, `ClimbingBall` should inherit `Ball` and then add the `speed` field). An UML scheme of such hierarchy could be the following:

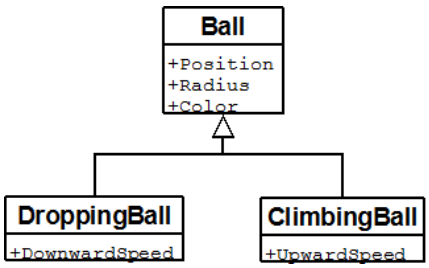


**Figure 7 - UML class hierarchy**

The predicates we identified while writing the inductive step of our abstract analysis (i.e. `touch`, `moveup`, `sameColor`, etc.) will become methods of the classes. For example, in our case study we identified the predicate `moveup`, which will become a method of the `ClimbingBall` class. This method will take a climbing ball as input and will return, as output, the same ball moved upwards of one time frame.

## 4.2 Operations to Queries

We will show how each expression of our game logic analysis can be easily translated using a functional paradigm. For brevity, we will not translate every single expression of the analysis, but only the most significant; however we will show the translation in more than one language (*C# + LINQ*, *F#*) and the general translation framework (the so called "comprehension notation").

### *4.2.1 Comprehension Notation*

*F#* is a multi-paradigm programming language, with a particular stress on its functional part (*F* stays for functional). The set-comprehension notation in this language can be schematized as the following:

```
seq {
        for iterator_name in iterated_sequence
        when predicate(iterator_name) = true
        -> f(iterator_name)
}
```

This syntax means that we take each element from `iterated_sequence` (calling it `iterator_name`), we check some conditions on it (`predicate`) and, if the conditions apply, we transform it using a function `f`. The value returned from that expression is a `Sequence`, which will be casted into a `Set`.

The other language we consider is *C#*: an object oriented language created as a mix of *Java*, *C++* and *Delphi*. Recently, *C#* integrated a library called *LINQ* (*Language Integrated Query*), which allows the user to manage collections of data (whether they are stored in local memory, in a remote database or in an XML file). In particular, *LINQ* offers a lot of utilities for managing lists (unfortunately, in *C#* we don't have the `Set` type as in *F#*, so we will use the `List` type). We show the list comprehension syntax in *LINQ* (you will notice that the expression is very similar to a *SQL* query):

```
        from iterator_name in iterated_sequence
        where predicate(iterator_name) = true
        select f(iterator_name);
```

This expression is almost identical to the one in *F#*: `from` corresponds to `for`, `where` to `when`, `select` to `->`. The two program slices are also very expressive: if you read them, you immediately understand what their effect will be.

In *LINQ* we can also use another notation, which stresses the *C#* object oriented flavour:

```
iterated_sequence
        .Where(iterator_name =>
                predicate(iterator_name))
        .Select(iterator_name => f(iterator_name))
```

The `Where` part of the expression keeps only the `iterated_sequence` elements for which the predicate returns true; then the `Select` part applies a function (`f`) to each of these filtered elements.

### *4.2.2 Sample Queries*

We can see the translation of a query in the various notations we explained above. For example, consider:

```
stuck = {m | m ∈ climbing ∧
           (∃s ∈ hanging | touch m s)}
```

The translation of this query in *F#* is the following:

```
let stuck = Set (seq {
        for m in climbing
        when Set.exists (fun s -> m.touch(s)) hanging
        -> m })
```

We iterate the climbing balls (`for m in climbing`) and we keep only those for which exists a hanging ball in contact with it (`Set.exists (fun s -> m.touch(s)) hanging`). Then we do not make any transformation on the results (using our general comprehension notation, we could say that `f` is the identity function).
What about *LINQ*? The same query becomes:

```
var stuck =
    climbing
        .Where(m =>
                hanging.Any(s => m.touch(s)));
```

We don't need to use the `Select` part of this notation, because we don't transform the elements, we only filter them. In the other *LINQ* notation (more *SQL-style*) we would write:

```
var stuck =
        from m in climbing
        where hanging.Any(s => m.touch(s))
        select m;
```

We see that the two queries (*F#* and *LINQ/SQL*) are almost twins (`from` ≈ `for`, `where` ≈ `when`, `Any` ≈ `exists`, `select` ≈ `->`). This similarity tells us that it is not important which specific language we use, as long as it offers a functional programming paradigm.

## 5. CONCLUSIONS AND FUTURE WORK

In this paper we shared our experience on formally analyzing the logic of some video game and then translating the analysis into functional code. The purpose of this technique is twofold: first of all, giving a method to easily (and correctly) implement small games; second, introducing students to functional programming in a way that is hands-on, engaging and not too much theoretical.

We tried this approach on a group of 6 CS students, and we made them develop games using this technique for their final undergraduate report. The results have been excellent: the students had no particular problem understanding how the technique worked and how to apply it to the games we assigned them. For simplicity we asked them to reproduce famous arcade videogames, so that they could focus on the technique and not on the design of the game or on the graphics (3D graphics is not needed): examples are Arkanoid, Pacman, Space Invaders, Asteroids, Risk, and so on. Some of the results are depicted in the following pictures:


**Figure 8 – Arkanoid and Risk**


**Figure 9 - Asteroids and PacMan**


**Figure 10 - Galaga and PacXon**


**Figure 11 - Puzzle Bobble**

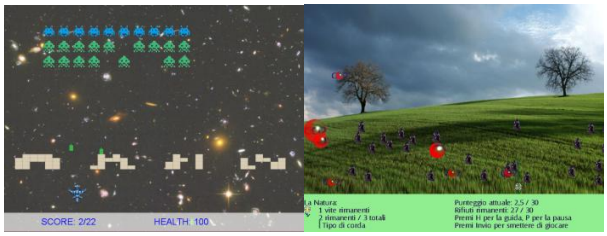


**Figure 12 - Space Invaders and SuperPang**

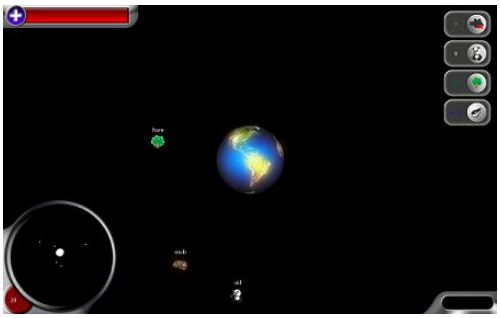

**Figure 13 - Project Solar System**

As it is clear from Figures 8 to 13, the games they produced are aesthetically appealing, and the resulting code quality was also very good (no bugs were found and the logic was seen as easily extendable). The time frame in which the games were produced was also surprising: each student developed three games in a period of approximately one month (while also taking classes, studying and having a normal social life). Most of the students were really pleased seeing the impact of mathematical concepts that too often feel a bit useless to a Computer Scientist: when they finished understanding our technique, the most common response was "Wow, that's what all that mathematics was for!".

The appreciation of the students for our technique is a boost to continue on this path. There are a lot of extensions that can be done to the technique. First of all, considering the entities more related to input. Secondly, we could consider the entire game as a FSM (finite state machine) with transitions between states (different phases of the game). Then, we could add some general support for FSM, so that the entities of the game could be FSM themselves, thus offering transitions and being more complex.

We will also start using this technique on a larger scale in full-blown programming courses, seen its potential for teaching complex topics and important coding lessons in an engaging and entertaining way.

## 6. REFERENCES


[1] Achten, P. 2008. Teaching functional programming with soccer-fun. In *Proceedings of the 2008 international Workshop on Functional and Declarative Programming in Education* (Victoria, BC, Canada, September 21 - 21, 2008). FDPE '08. ACM, New York, NY, 61-72. DOI= http://doi.acm.org/10.1145/1411260.1411270

[2] Box, D. and Hejlsberg, A. 2007. LINQ: .NET Language Integrated Query. http://msdn.microsoft.com/en-us/library/bb308959.aspx.

[3] Chakravarty, M. M. and Keller, G. 2004. The risks and benefits of teaching purely functional programming in first year. *J. Funct. Program.* 14, 1 (Jan. 2004), 113-123. DOI= http://dx.doi.org/10.1017/S0956796803004805

[4] Costantini, G., Maggiore, G. and Cortesi, A. 2009. Learning by fixing and extending games. In *Proceedings of Eurographics* 2009 (Muenchen, Germany, March 30 – 3 April, 2009)

[5] Harrison, R. 1993. The use of functional languages in teaching computer science. In *Journal of Functional Programming* (1993), 3:67-75 Cambridge University Press.

[6] Hughes, J. 2008. Experiences from teaching functional programming at Chalmers. *SIGPLAN Not.* 43, 11 (Nov. 2008), 77-80. DOI= http://doi.acm.org/10.1145/1480828.1480845

[7] Jones, S. P. and Wadler, P. 2007. Comprehensive comprehensions. In *Proceedings of the ACM SIGPLAN Workshop on Haskell Workshop* (Freiburg, Germany, September 30 - 30, 2007). Haskell '07. ACM, New York, NY, 61-72. DOI= http://doi.acm.org/10.1145/1291201.1291209

[8] Joy, M., Matthews, S. 1994. Some experiences in teaching functional programming. In *International Journal of Mathematical Education in Science and Technology*, 25 (2), 165-172.

[9] Meijer, E., Beckman, B., and Bierman, G. 2006. LINQ: reconciling object, relations and XML in the .NET framework. In *Proceedings of the 2006 ACM SIGMOD international Conference on Management of Data* (Chicago, IL, USA, June 27 - 29, 2006). SIGMOD '06. ACM, New York, NY, 706-706. DOI= http://doi.acm.org/10.1145/1142473.1142552

[10] Pickering, R. 2007 Foundations of F#. Apress.

[11] Thompson, S. J. 1997. Where Do I Begin? A Problem Solving Approach in teaching Functional Programming. In *Proceedings of the9th international Symposium on Programming Languages: Implementations, Logics, and Programs: including A Special Trach on Declarative Programming Languages in Education* (September 03 - 05, 1997). H. Glaser, P. H. Hartel, and H. Kuchen, Eds. Lecture Notes In Computer Science, vol. 1292. Springer-Verlag, London, 323-334.