Functional Paradigm in Embedded Systems

Anita Szabó[1], Norbert Sram[2]

[1]Subotica Tech, Serbia, saboanita@vts.su.ac.rs
[2]Inovacioni Centar D.O.O., Subotica, Serbia, norbert.schramm@gmail.com

Abstract: Embedded computer systems are rapidly gaining more and more ground, from stand alone systems to networked based systems with complex logic. The primary goal of embedded systems is to provide a reliable service over a period of time without any kind of intervention. These systems are mainly programmed in low level languages and are often the subject of software flaws inherited from unsafeness of these languages. The purpose of our research is to develop safe, secure, verifiable software for these systems, without significant performance loss. With the use of functional paradigm we are able to rapidly develop elegant code and solve complex tasks with ease.

Keywords: embedded systems, functional programming

1 Introduction

The development process of embedded systems is a slow and costly process due to its low level nature. Creating high-end, reliable systems with the traditional methods is a process which involves a great deal of work. To simplify this process we need to use tools, which provide elegant solutions. One of the main usages of embedded systems is controlling. Controllers vary a great deal. Low ended controllers usually monitor a sensor, based on the results make a decision and execute a command or sequence of commands. On the other hand, high end controllers have numerous tasks to do. They monitor and coordinate simple controllers. Their decision making can involve complex logic. Because of these requirements the complexity of these systems makes it even harder to develop them.

Functional programming languages, such as OCaml, Haskell, Clean have been used to create desktop and server side applications as well, with great success. There are projects [2, 7, 8] which try to adopt functional programming in embedded systems by creating custom designed languages. It has been shown that soft computing algorithms can be used in functional programming languages as well [1, 13, 14, 15, 16].

By using various examples, and describing how they can be written in a functional style, we will show that:

- Functional programming simplifies software development, including embedded development as well.
- Existing, general purpose functional programming languages can be adopted for embedded development.
- Trade off between performance and safety, clarity, and verifiability is more than acceptable.
- The use of soft computing algorithms simplifies the software complexity and increases flexibility.

The functional notation used in this paper is based on OCaml [3] an ML derivate.

Section II provides a brief overview of functional programming. Section III lists some of the advantages of functional style. Section IV describes the languages and tools proposed and used by the authors. Section V, VI and VII, discuss soft computing, one of it's branches – complex systems - and present an example to explore the use and advantages of functional programming languages to implement solutions for real world problems. Section VIII contains some concluding remarks. Appendix A provides the original sources.

2 A Brief Overview of Functional Programming

For a more detailed examination of functional programming see [4]. Functional programming is a programming paradigm that treats computation as the evaluation of mathematical functions and avoids state and mutable data. It emphasizes the application of functions, in contrast with the imperative programming style that emphasizes changes in state. One of the great advantages of functional programming is that sub-objects are entirely independent, and may therefore be computed in any order.

3 Advantages of Functional Programming Languages

In functional programming languages, computation is largely performed by applying functions to values. This means that the value of an expression depends only on the values of its sub-expressions (if any). Thus the evaluation does not produce side effects. The value of an expression cannot change over time. This

means that there is no notation of state. Computation may generate new values, but not change existing ones. By using functional paradigm we achieve:

Simplicity – no explicit manipulation of memory. Values are independent of underlying machine with assignments and storage allocation. Garbage collection.

Power – recursion for iteration. Functions are first class values, meaning that they can be used as values for expressions, passed as arguments, placed in data structures. Functions need not have a name.

Conciseness - functional programs tend to be a fraction of the size of imperative programs, this means that we have less code to read, less to maintain and less to debug.

More Reliable - functional program more closely resemble the specification, are more understandable, and tend to work the first time. Values are always internally consistent. Subtle bugs are rare.

Persistence - we always have access to previous versions of the data from which to compare with the more recent data.

Optimization - There are more opportunities to eliminate infrastructure taxes, allowing the computer to do more of the work that traditionally programmers would do by hand.

4 Functional Programming In Practice

The number of Linux enabled devices is rapidly increasing. This opens the door to new possibilities in embedded development. Development is no longer tied to hardware. Embedded applications should also use the operating system as their interface to the hardware.

The used tools are required to support cross platform development. Including embedded hardware as well in case of native-code compilers. Since embedded systems offer limited resources we must take into consideration the efficiency of the run-time systems as well. Based on the requirements, the authors choose Linux as their operating system. OCaml and F# as the development platform.

OCaml [3] is a general-purpose programming language, designed with program safety and reliability in mind. It supports functional, imperative, and object-oriented programming styles. The OCaml system contains a high-performance native-code compiler for numerous processor architectures, as well as a bytecode compiler and an interactive read-eval-print loop for quick development and portability. The Objective Caml distribution includes a comprehensive standard library, a replay debugger, lexer and parser generators, a pre-processor pretty-printer and a documentation generator. This makes it an excellent a productive

environment. The run-time environment of the language is small and fast with moderate memory usage.

F# [5] is a functional and object-oriented programming language for the CLI (Common Language Infrastructure). Strength of F# is its setting within .NET framework. It provides access to numerous platforms, through the Mono [6] project. The vast amount of libraries make it an excellent tool for real-life projects. Also it has a cross-compiling compatible core with OCaml. The performance of F# programs is mostly dependent on the virtual machine it runs in.

Both languages have similar features, which include:

Strongly typed – The types of all values are checked to make sure that they are used appropriately**.** Any inappropriate use (a *t*ype mismatch) incurs an error. Strong typing improves program reliability and reduces development time by highlighting type mismatches. This is an important part of safe programming.

Statically typed – All typecheking is done entirely at compile time. This means that all type errors are detected before the program is run. An important advantage of static type checking is that run-time type checks are no longer necessary and can be removed, greatly improving program performance.

Type inference – Most conventional statically-typed programming languages (e.g. C, C++, C#, Java) require the programmer to repeatedly restate the types of expressions. Restating types bloats the source code of a program, making it more difficult to navigate, develop and maintain. In type inferred languages it is never necessary to explicitly declare types, however, defining important types in a program is a good way to leverage static type checking by providing machine-checked documentation, improving error reporting and tightening the type system to catch more errors. Type inference is based on Hindley–Milner or Damas–Milner algorithm [11].

5 Functional Programming in an Imperative Environment

Fucntional programming languages are designed from ground up to support functional paradigm,but the usage of functional paradigm and style can be incorporated into standard imperativ languages as well. Using functional paradigm in the development of embedded software produces high quality software. By eleminating uneccessary states in the software, it will become easiser to maintain and also easier to support cuncurrent execution of the software. Concurrency is an important feature for embedded systems as well due to the advancements in embedded hardware architectures. Also the constraints of embedded hardware require a special approach for efficient support of concurrent execution. Using a

stateless functional implementation we can model our concurrent software as a complex system of execution nodes. The execution nodes have a type which for a given input type ’a’ creates a give input type ’b’. This operation will always give the same result for the same input. The execution nodes are referentialy transparent, they do not support destructive actions. The type variables ’a’ and ’b’ can have the same type as well.

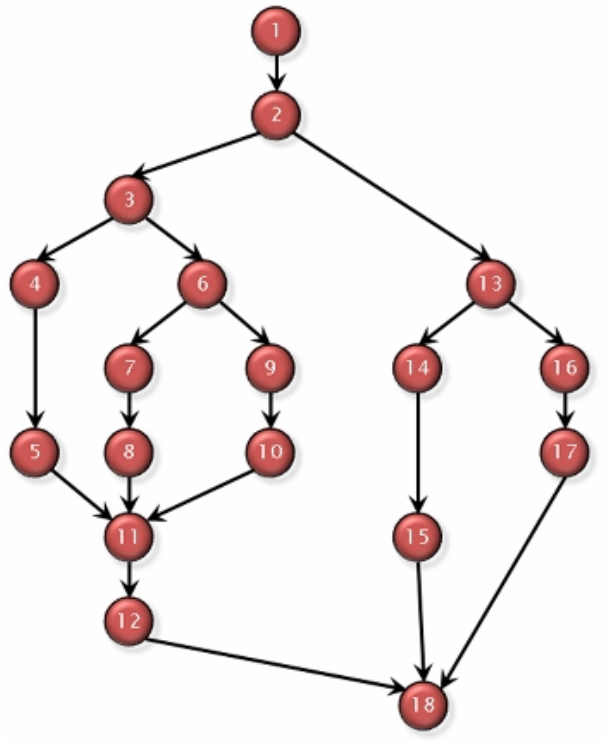

Figure 1
A system of execution nodes

Viewing the architecture of the embedded software as a systems of execution nodes we can study how relationships between parts give rise to the collective behaviors of a system and how the system interacts and forms relationships with its environment [9, 10]. This kind of approach to the development of embedded software does not reuqire the usage of a functional progarmmin language only the usage of the functional programming idom, which can be done in an imperative language as well.

6 Complex Systems

Complex systems are always a network of some kind. Often embedded solutions are networks as well. The modeling of complex systems can be achieved with the help of creating cellular automata. A cellular automaton is a discrete model. It consists of a regular grid of cells, each in one of a finite number of states. The grid can be in any finite number of dimensions. Time is also discrete. To show, why functional programming excels at working with complex systems, we will implement the best-known cellular automaton, the Game of Life. It has been devised by the British mathematician John Horton Conway in 1970 [12]. The game doesn't actually require input from the human player, only an initial state. One interacts with the Game of Life by creating an initial configuration and observes how it evolves. The rules are simple. The universe of the Game of Life is an infinite two-dimensional orthogonal grid of square cells, each of which is in one of two possible states, live or dead. Every cell interacts with its eight neighbours, which are the cells that are directly horizontally, vertically, or diagonally adjacent. At each step, the following transitions occur.

- Any live cell with fewer than two live neighbours dies, as if by loneliness.
- Any live cell with more than three live neighbours dies, as if by overcrowding.
- Any live cell with two or three live neighbours lives, unchanged, to the next generation.
- Any dead cell with exactly three live neighbours comes to life.

The initial pattern constitutes the 'seed' of the system. The first generation is created by applying the rules simultaneously to every cell in the seed.

From a theoretical point of view, it is interesting because it has the power of a universal Turing machine: that is, anything that can be computed algorithmically can be computed within Conway's Game of Life.

In the following section we will explain parts of the implementation where functional style excels and elaborate the provided advantages. For full source code see Appendix A.

The first step is to specify the size of the grid (the universe) and create it. Next, we define our initial pattern (seed). For the sake of simplicity we will use a fixed grid size in our example.

```
let size = 10 ;;

let the_grid = Array.create_matrix size size 0  ;;

let the_pattern =

 [|
```

```
  [|1; 1; 1|];
  [|1; 0; 0|];
  [|0; 1; 0|]
|] ;;
```

Because we are using a matrix as our grid, we need to provide the functionality to iterate through a matrix. In imperative languages, we would achieve this with iteration constructs, such as the for loop. In functional languages we can get the same result using higher-order functions. Functions are higher-order when they can take other functions as arguments. Higher-order functions are a feature of functional programming languages, where functions are first-class values. The phrase "first-class" is a computer science term that describes programming language entities that have no restriction on their use.

```
let iteri_matrix f = Array.mapi (fun i -> Array.mapi (f i)) ;;
val iteri_matrix :
(int -> int -> 'a -> 'b) -> 'a array array -> 'b array array = <fun>
```

The type of the matrix iteration function shows that we created a generic function for iterating through a matrix. It is a function, which takes a function as its input and returns a function. The main advantage of this approach and functional languages is reusability. It is simple and generic.

Our next step is to provide functionality for adding a pattern to the grid. First we define a function to get a pattern cell at a particular coordinate. It returns 0 if the coordinates are out of range, because we want to use it to sum the pattern and the grid without having to make the pattern matrix the same size as the grid matrix.

```
let get_pattern_cell = function
   (i, j) when i < 0 || j < 0 -> 0
 | (i, j) when i < Array.length the_pattern &&
              j < Array.length the_pattern ->   the_pattern.(i).(j)
 | _ -> 0 ;;
```

Instead of the traditional if-else branching solution, we used pattern matching with guards. This is a feature used in functional languages. Pattern matching provides the ability to process data based on its structure. The input of the 'get_pattern_cell' function is a tuple, which is destructed by pattern matching. The destructed data is check for the condition presented after the 'when' clause.

```
let add_pattern_to_grid grid =
  let offset = (size - Array.length the_pattern)
    iteri_matrix(fun i j _ -> grid.(i).(j) + get_pattern_cell(i - offset, j -  offset)) grid ;;
```

The above function adds a pattern to the center of the grid using the previously defined matrix iteration function. This implementation uses function composition which is essential in functional style.

We need to extract to identify the neighbours of a cell, to run the game rules on them. Coordinates of the cell are passed to this function as tuples. Tuples provide the ability to group data together, without the burden of creating a new type.

```
let neighbours (i, j) =
 let prev i = if i = 0 then size - 1 else i - 1 in
 let next i = if i = size - 1 then 0 else i + 1 in
  [
   (prev i, prev j);
   (prev i, j);
   (prev i, next j);
   (i, prev j);
   (i, next j);
   (next i, prev j);
   (next i, j);
   (next i, next j)
  ]
```

We used the above function to retrieve the sum of live neighbours. The implementation of the sum function is based on higher order functions. We pass the operator + as function, which will be applied to all the elements of the list.

```
let cell_add_neighbours (i,j) grid =
 let rec sum  = List.fold_left ( + ) 0 in
  sum (List.map ( fun(a,b) -> grid.(a).(b))( neighbours (i,j))) ;;
```

To get the neighbours sum for all cells, we use the function composition technique.

```
let add_neighbours grid =
 iteri_matrix (fun i j _ -> cell_add_neighbours (i,j) grid) grid ;;
```

Finally, we need to run the game rules on each cell. Pattern maching makes the implementation of the rule set extremly simple. It allows us to list a set of rules for matching each element in a tuple without writing loads of conditional logic.

```
let cell_live_or_die cellvalue neighboursum =
 match (cellvalue, neighboursum) with
   (1, (2 | 3)) -> 1
```

```
| (0, 3) -> 1
| (_, _) -> 0 ;;
```

We use our matrix iteration function to test every cell in the gird.

```
let live_or_die grid neighbour_sum_grid =
 iteri_matrix (fun i j _ -> cell_live_or_die grid.(i).(j)  neighbour_sum_grid.(i).(j)) grid ;;
```

Below is the output of the game with the initial pattern, which ran for 10 generations. Result of some generations is ommitted. The pattern is the famous 'Glider'.

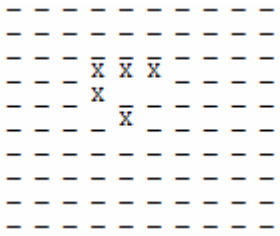

Figure 2
First generation

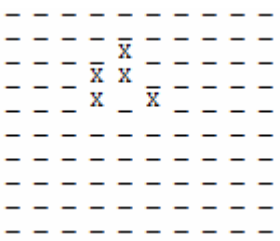

Figure 3
Second generation

Left out a few generations here.

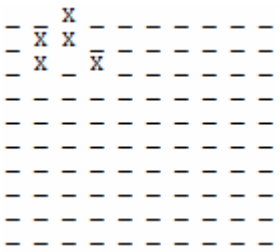

Figure 4
Ninth generation

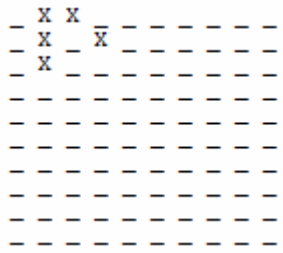

Figure 5
Tenth generation

Conclusion

We have explored the possibility of the use of functional programming in domains which are dominated by imperative, data driven approaches implemented in low level languages. We presented a simple example which shows that, with an output-driven functional approach it is possible create software, which is made up of simple and reusable components. We have shown that there are tools for using this technique in practice, even in embedded systems. The presented sample program was executed on Marvell's (formely Intel's) XScale PXA270 processor using Debian Etch. The tests were cunducted using OCaml's bytecode compiler. It was also tested with F# under Mono.

This approach led us to start the development of an efficient and generic framework used for concurrent execution in embedded systems. Work is currently being done on a practical implementation.

Appendix A

The full source of the Game Of Life implementation.

```
let size = 10 ;;
let the_grid = Array.create_matrix size size 0  ;;
let the_pattern =
  [|
    [|1; 1; 1|];
    [|1; 0; 0|];
    [|0; 1; 0|]
  |] ;;
let iteri_matrix f = Array.mapi (fun i -> Array.mapi (f i)) ;;
let get_pattern_cell = function
    (i, j) when i < 0 || j < 0 -> 0
  | (i, j) when i < Array.length the_pattern &&
      j < Array.length the_pattern -> the_pattern.(i).(j)
  | _ -> 0 ;;
let add_pattern_to_grid grid =
```

```
  let offset = (size - Array.length the_pattern) / 2 in
    iteri_matrix(fun i j _ -> grid.(i).(j) + get_pattern_cell(i - offset, j - offset)) grid ;;
let neighbours (i, j) =
  let prev i = if i = 0 then size - 1 else i - 1 in
  let next i = if i = size - 1 then 0 else i + 1 in
    [
      (prev i, prev j);
      (prev i, j);
      (prev i, next j);
      (i, prev j);
      (i, next j);
      (next i, prev j);
      (next i, j);
      (next i, next j)
    ]
let cell_add_neighbours (i,j) grid =
  let rec sum  = List.fold_left ( + ) 0 in
    sum (List.map ( fun(a,b) -> grid.(a).(b))( neighbours (i,j))) ;;
let add_neighbours grid = iteri_matrix (fun i j _ -> cell_add_neighbours (i,j) grid) grid ;;
let cell_live_or_die cellvalue neighboursum =
  match (cellvalue, neighboursum) with
      (1, (2 | 3)) -> 1
    | (0, 3) -> 1
    | (_, _) -> 0 ;;
let live_or_die grid neighbour_sum_grid =
  iteri_matrix (fun i j _ -> cell_live_or_die grid.(i).(j) neighbour_sum_grid.(i).(j)) grid ;;
let print_cell cell =
  if cell = 1 then Printf.printf("X ") else Printf.printf("_ ") ;;
let print_grid grid =
  Array.iter (fun line -> Array.iter print_cell line) grid;
  Printf.printf "\n";
  Printf.printf "\n\n" ;;
let rec do_generations n grid =
  print_grid grid;
  match n with
      0 -> Printf.printf "end\n"
    | _ -> do_generations (n - 1) (live_or_die grid (add_neighbours grid)) ;;
let _ = do_generations 10 (add_pattern_to_grid the_grid) ;;
```

References

[1] Gary Meehan. Fuzzy Functional Programming. University of Warwick, 1997

[2] University of St Andrews. The Hume Programming Language. WWW: http://www-fp.cs.st-andrews.ac.uk/hume/index.shtml

[3] Xavier Leroy. The Objective Caml system. INRIA 2007

[4] Simon Thompson. Haskell: The Craft of Functional Programming 2nd Edition. Addison Wesley, 1999

[5] Microsoft Research. F#. WWW: http://research.microsoft.com/fsharp/fsharp.aspx

[6] Mark Mamone. Practical Mono. Apress 2005

[7] Kevin Hammond. Is it Time for Real-Time Functional Programming. Trends in Functional Programming 4, Intellect. 2005

[8] Kevin Hammond, Gudmund Grov, Greg Michaelson, and Andrew Ireland. Low-Level Programming in Hume: an Exploration of the HW-Hume Level. Implementation of Functional Languages. *2006*

[9] Stephen Wolfram. Computer Software in Science and Mathematics. Scientific American. 1984

[10] Stephen Wolfram. Cellular automata as models of complexity. Nature 311. 1984

[11] Luis Damas and Robin Milner. Principal Type-Schemes for Functional Programs. Annual Symposium on Principles of Programming Languages. 1982

[12] Martin Gardner. Mathematical Games: The fantastic combinations of John Conway's new solitaire game ‘life’. Scientific American. 1970

[13] Lotfi A. Zadeh. Soft Computing and Fuzzy Logic. IEEE Software, Volume 11, Issue 6. 1994

[14] Márta Takács. Approximate Reasoning in Fuzzy Systems Based On Pseudo-Analysis. Phd Thesis, Univ. of Novi Sad. 2004

[15] Márta Takács. Critical Analysis of Various Known Methods for Approximate Reasoning in Fuzzy Logic Control. International Symposium of Hungarian Researchers on Computational Intelligence. 2004

[16] Mamdani E., H., Gaines,B.. Fuzzy Reasoning and Its Applications. Academic Press, New York. 1981