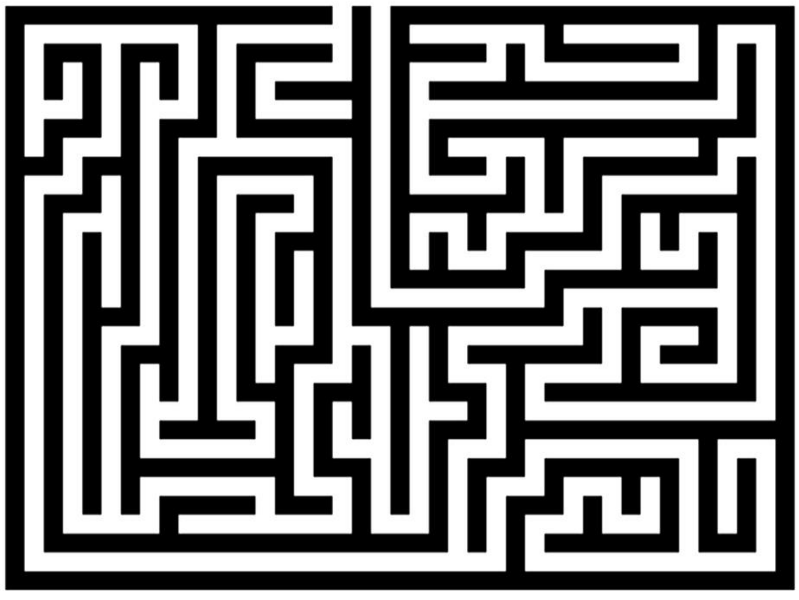
MAZE SOLVER

A NONDETERMINISTIC COMPUTING APPROACH TO MAZE SOLVING



DECEMBER 15, 2015

MARKOUS SOLIMAN & MOUSTAFA ELSHAABINY

Table of Contents

1. Introduction

This report is intended to provide a brief overview of functional programming and its core concepts, a high-level explanation of nondeterministic computing, and, finally, to showcase an interesting real-world application to demonstrate the effectiveness of the concepts mentioned above. In our report, we assume a basic understanding of programming and software design terminology. However, our intention is to shine a light onto the functional programming arena in a reader-friendly manner.

Although it can be daunting, functional programming is one of the most powerful and respected programming paradigms nowadays. In fact, it is considered one of the key players in the "big data" revolution[1]. Essentially, functional programming is about adopting a programming style and a set of ideas that break down code into smaller pieces, which can be easily debugged and reused. The two main advantages that distinguish functional programming from other programming styles are immutability and statelessness. Simply put, immutability allows software engineers to write cleaner code, better abstractions, and, more interestingly, concurrent programs. Immutable is not the same thing as unchangeable, however. In functional programming, we create new data structures to store the changed data in existing ones rather than overwriting it. On the other hand, the stateless nature of functional programs prevents any previous knowledge from interfering with programs' execution. In other words, stateless programs execute every task as if it is the first time. This nature allows functions to operate in a 'vacuum' without relying on outside values to do their calculations, and a function call can have no effect other than to compute its results. In order to perform tasks, those programs only use the

[1] https://ascent.atos.net/the-rise-of-functional-programming/

parameters being passed to them (later on, we illustrate how we utilized this feature in our Maze Solver).

Those features mentioned earlier motivate the idea of distributed systems. With the current trends shifting to computing with more, rather than faster, processors, giant companies like, Google, Microsoft, IBM, and Facebook, now run their programs in data centers that are accessed through the cloud. Very soon, most companies will shift to cloud computing, too. Thus, developing software for these distributed environments becomes essential and learning functional programming becomes a must.

2. Project Scope

When we discussed nondeterministic computing as a potential project area, we, for some reasons, thought of being lost in a maze and presented by several options not knowing which one leads to the correct path to solving it. Think about this for a second. What would you do? What would a Scheme program do? Well, we designed a nondeterministic computing system that finds the correct path to solving any maze in a couple of seconds. We will demonstrate how exactly the system does that in subsequent sections of this report.

The essence of nondeterministic computing is *automatic search* that is invaluable to "generate and test" types of applications. The maze problem is one such application. Moreover, we use bounded non-determinism, where every process is confined to a finite number of choices (You should not be stuck in a maze with an infinite number of directions you can take!). In the next section, we describe some of the methods and operations that the nondeterministic approach utilizes.

3. Methods and Operations

3.1. *amb* Evaluator

The idea of *amb* is difficult to understand, at first, but here we will try to give a high-level description of what it is and simplify things a little bit. *amb* is a nondeterministic operator that takes zero or more expressions and makes an "ambiguous" (or nondeterministic) choice among them. Those expressions represent the possible values of the *amb* expression. *amb*'s choice does not guarantee a successful return value that is accepted by our program, however. *amb* should always be restricted by a set of rules that makes the results of certain choices invalid. When the result of a choice is invalid, amb uses a sleek backtracking mechanism (called call-with-current-continuation) to try other possible values until it finds one that returns a valid result. Should amb's choice cause final failure, the program will backtrack to the chronologically previous amb call. In a sense, the restrictions that we impose on amb in our programs veto amb's choice to what we want it to have picked from the beginning.

There are two types of non-determinism: *weak and strong non-determinism*[2]*.* The implementation of amb that we use in this project falls under the weak non-determinism umbrella[3], where its choice follows a certain order and is not "fully" nondeterministic. In other words, we should be able to expect amb to choose the first expression's value first followed by the second expression's value, and so on. As long as it fails our assertions, amb will try the next possible choice. Eventually, it will make a sequence of correct choices that ultimately returns the desired results. Since this backtracking happens behind the scenes, it may seem like amb is always making the correct choices. It might be tempting to think that amb has

[2] http://comjnl.oxfordjournals.org/content/35/5/514.full.pdf
[3] we used the macro implementation of amb

super powers, but we are here to tell you that everyone makes mistakes, even amb! The only difference is that amb “knows” where it made that mistake and can “travel” back to that point in time and try a different choice.

3.2. Continuation Passing

Current continuation and continuation passing plays a vital role in amb, namely, in backtracking, so it is important to explain what they mean and how they work to fully understand amb. Here’s a quote *from Chapter 3, The Scheme Programming Language Book* that serves out demonstrations purposes, “During the evaluation of a Scheme expression, the implementation must keep track of two things: (1) what to evaluate and (2) what to do with the value”[4]. Here is an example in pseudo-code, assume we want to evaluate the following expression: if x is *true* do y, otherwise do z. “What to evaluate” is if x is *true* and "what to do with the value" is to make the decision which of y or z to do/evaluate *and* to do so. In functional programming (and Scheme specifically) the “what to do with the value” is called the *continuation* of a computation. Amb uses continuations as a formal way of describing what is left to do in the computation. The implementation of amb that we used in our project uses Scheme’s call-with-current-continuation (also known as call/cc) operator to manipulate the backtracking mechanism.

[4] http://www.scheme.com/tspl3/further.html

4. Maze Solver

4.1. *High-level description*

Our Maze Solver program allows users to input the dimensions of the maze they would like to create. For example, a user can enter any two positive integers. Our program then uses recursive division to generate a random maze. Before we explain what recursive division is and how it works to design our mazes, let us first talk about how we represent a maze in Scheme. In Figure 1 we illustrate how we converted a typical maze into our Scheme-

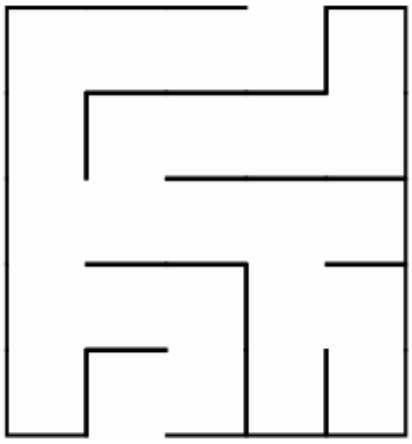


Figure 1.a – Typical maze

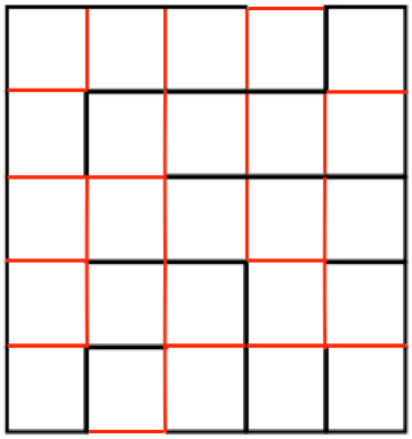


Figure 1.b – Cellular

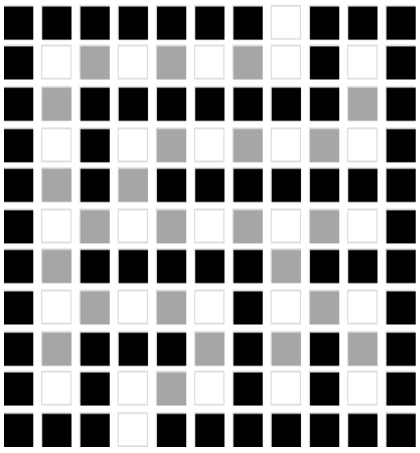


Figure 1.c – Visualized scheme representation

```
((1 1 1 1 1 1 1 0 1 1 1)
 (1 0 2 0 2 0 2 0 1 0 1)
 (1 2 1 1 1 1 1 1 1 2 1)
 (1 0 1 0 2 0 2 0 2 0 1)
 (1 2 1 2 1 1 1 1 1 1 1)
 (1 0 2 0 2 0 2 0 2 0 1)
 (1 2 1 1 1 1 1 2 1 1 1)
 (1 0 2 0 2 0 1 0 2 0 1)
 (1 2 1 1 1 2 1 2 1 2 1)
 (1 0 1 0 2 0 1 0 1 0 1)
 (1 1 1 0 1 1 1 1 1 1 1))
```

Figure 1.d – Typical scheme representation

In order to represent the maze in Figure 1.a in Scheme we have to first think of it as if it as a “cellular grid”. The lines in Figure 1.b should in fact be cells if we want to represent that maze in Scheme. You can see that in Figure 1.c. All black lines represent a “wall” in the maze and all red lines represent an “open gate”. We give all walls a representative value equal to 1 and all gates a value equal to 2. Lastly, the white cells represent a tile in the maze. We give all white cells a representative value equal to 0. As you can see, such representation results in a $(2n+1) \times (2m+1)$ Scheme-maze.

There are four directions the non-deterministic program (we will call it player from now on) could take, namely forward, backward, right or left. The current position of the player gets passed to the direction’s function, which “moves” the player. The position of the player is represented in a (row_index, column_index) pair. A move to the right means that new position is (row_index, column_index +1), left is (row_index, column_index -1), forward is (row_index+1, column_index) and backward is (row_index-1, column_index). Those four directions are amb’s finite list of choices as we introduced in Section 2 of this document.

Next we introduce our game rules and restrictions. Each position (row_index, column_index) has a value associated with it. As we mentioned earlier, this value could be either 0, 1 or 2. If the player makes a move into a position that has a value 0 or 2 that means it is a valid move and we can update the player’s current position to this new position. Then we store it into a list we call correct path. Otherwise, it is an invalid move and the player has to go back and make another choice. Also, if the player’s current position is equal to the starting point of the maze, then the player should not move backward. In other words, a backward move in this case is invalid and would result in failure of the current program. Our last rule ensures that

when the player chooses a direction for their next move, the new position is not equal to his previous position. This is to make sure that the player is always advancing.

What if the player made a sequence of choices that led him to a dead-end? Consider this case below.

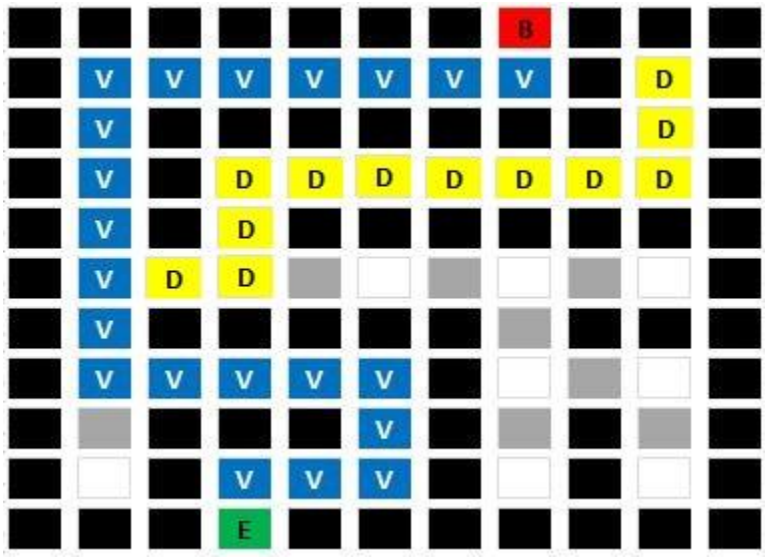


Figure 2 – Example of a maze with a successful and dead-end paths

The player can neither move right, left, forward, nor backward. The player cannot move backward is due to our restrictions which state that a player's new position cannot be the same as their previous one. Of course, the player cannot move in the other directions because of our first rule, which states that a player cannot move into a cell that has a value of 1 (because you cannot move into a wall!). That means the player has exhausted all options and is now stuck. Furthermore, that means that our current correct path list does not have the correct path to solving the matrix. Well, that is only half true. The correct path list had the correct path up until a point where the player made a valid, yet wrong decision that led to this dead-end. That is why we came up with a new way of solving this problem. We allow the player to change the value of his current position if he found that the path they chose led to a

dead-end. In that case, we pop the player's current position from the correct path list and assign their previous correct position (cadr of correct path list) to be their current position and we hope that this would be sufficient to get the player out of that dead-end. If it was not, then we repeat that backtracking process until the player is out of that dead-end. With those rules and restrictions, a player can solve any given Scheme-maze.

4.2. Details of implementation

assert

As mentioned in the previous section, some rules to the game have to be applied. These rules are implemented through "assert" function. This function, as shown in the following figure, is not part of the amb-macro given, but it's one of the functions that use amb to retain the backtracking policy.

"assert" function takes a predicate as its only parameter and returns the call to the *amb* operator if the predicate is not satisfied. By now, we already know that calling the *amb* operator will backtrack the current decision, because it apparently is wrong, and makes another decision from the list, hoping that it leads to a better solution.

```
(define assert
  (lambda (pred)
    (if (not pred) (amb))))
```

Figure 3 – assert function (scheme)

an-element-of

It's another ready function that is implemented to serve the amb operator in the best possible and reusable way. It basically takes a list of elements (it could be a list of atoms or a list of lists or even more complicated form of lists) and it chooses one element at a

time in a sequential order. It uses the amb operator to be able to keep track of the position of the current element it called and call the next element, if that element didn't satisfy all the rules of the game. But wait! What will happen if the player is at the last element of that list? Then amb has exhausted all the possible successful attempts to have this program works and has reached to the final failure state, where it sadly displays "amb tree exhausted".

```
(define (an-element-of items)
  (assert (not (null? items)))
  (amb (car items) (an-element-of (cdr items))))
```

Figure 4 – an-element-of function (scheme)

Is there a solution to that problem? Of course, there is. Our solution lies under the magic hat "iterations-start-at"

Iterations-start-at

This function also uses the amb operator. In fact, this function is a very optimistic one, where it only looks at the previous state and increments the value of the given parameter by 1. In other words, this function serves as an unbounded function, where it keeps the number of iterations increasing as it gets called.

But in what way would we use such a function? Actually, this function keeps the program alive every time "*an-element-of*" function runs out of options. We can get more insight about this through explaining the game itself.

```
(define (iterations-start-at n)
  (amb n (iterations-start-at (+ n 1))))
```

Figure 5 – iterations-start-at function (scheme)

maze-game

Our game function starts with couple definitions that are required before we start working with amb operators and call/cc. Firstly, we assign a random point of the top border of the maze to be the begin point. Similarly, we assign an end point in the bottom border. Also, we set the begin point to be the current position as a start of the game, as well as the previous position to be the origin of the maze (0,0). Lastly, before we start working with amb, we will define the correct-path as a list of the current position and the previous position.

We start the first amb operator with the unbounded amb function “iterations-start-at” that will only break if the current position of the player is equal to the end point, which is the end of the maze. This is done through the last “assert” statement we have in the function. Also, without satisfying this condition, the program will keep running, trying to reach to a successful state.

Once we set the first amb function, we then have to make sure to call “directions-list” function which shuffles the list of directions we have declared in a previous part of the file, to keep the program more non-deterministic. Then, we assign our second amb function, using “an-element-of” which takes the list of directions, to a variable “next-move” which keeps track of the next-move the player makes in the maze every time it gets called. We also have to keep track of the current position and the previous one, by setting them up to the (car) and the (cadr) of the correct-path.

Keeping track of the correct-path in every call to the amb functions is very important, as it tells the player on which moves are valid and which are invalid that require backtracking. Through couple assert functions, we make sure that the player never goes out of the maze

from the begin point by taking a backward step, we make sure the next step value is not equal to 1, which is a wall, and finally, we make sure that the next step position is not equal to the previous position so it doesn't go back on what was already processed. With all these assert functions and by the end of the function, it reaches to the last assert to make sure it's successful and returns back with the correct path it took to complete the task.

```
(define (maze-game maze)
  (define begin-point (find-begin-point maze))
  (define end-point (find-end-point maze))
  (set! current-position begin-point)
  (define previous-position '(0 0))
  (define correct-path (list current-position previous-position))

  (let ((num-iterations (iterations-start-at 1)))
    (set! directions-list game-directions)
    (let ((next-move (an-element-of directions-list)))
      (set! current-position (car correct-path))
      (set! previous-position (cadr correct-path))
      (let ((next-move-position (new-position current-position next-move)))
        (set! options (cons next-move options))
        (if (exhausted-all? options)
            (begin (set! correct-path (cdr correct-path))
                   (set! maze (change-value-in-maze maze current-position 1))
                   (set! options '())))
        (assert (not (and (eq? next-move 'backward) (equal? current-position begin-point))))
        (assert (not (= (value-of-position next-move-position maze) 1)))
        (assert (not (same-position? next-move-position previous-position)))
        (set! correct-path (append (list next-move-position) correct-path))
        (set! previous-position current-position)
        (set! current-position next-move-position)
        (set! options '())))
    (assert (equal? current-position end-point))
    (display "Number of iterations: ")
    (display num-iterations)
    (newline)
    (display "Correct path: ")
    (set! correct-path (append (cdr (reverse correct-path)) (list end-point)))
    (display correct-path)
    (newline)
    (display 'DONE)
    (newline)
    correct-path))
```

Figure 6 – maze-game (scheme)

5. System Improvements

We also investigated the possibility of, instead of pushing the correct path onto a list, using nested calls of amb to better utilize the call-with-current-continuation. It would have been a better functionally programming style compared to resetting the value of the state variables directly. However, since we generate random mazes each time Maze Solver runs, we do not know the exact number of steps required to solve that given maze. Therefore, not only that would result in nesting many amb calls within each other, but present another logical puzzle that we would need to do more research on.

6. Conclusion

In conclusion, Maze Solver is just one application of some powerful functional programming paradigms. Maze Solver is not perfect, and there are areas where we can improve it. It is intended to be purely educational and experimental and not production. On a different note, learning functional programming and applying to our Maze Solver has been a fun experience. Over the course of this semester, our team developed a love-hate relationship with functional programming, mainly due to the simple fact that it challenged every assumption we had about writing software. Ultimately, we hope this report will inspire programmers to get out of their comfortable --Object-Oriented Programming-- zone to explore new programming paradigms.