

MAZE SOLVER

A NONDETERMINISTIC COMPUTING APPROACH TO MAZE SOLVING



December 15, 2015

**Markous Soliman & Moustafa Elshaabiny**

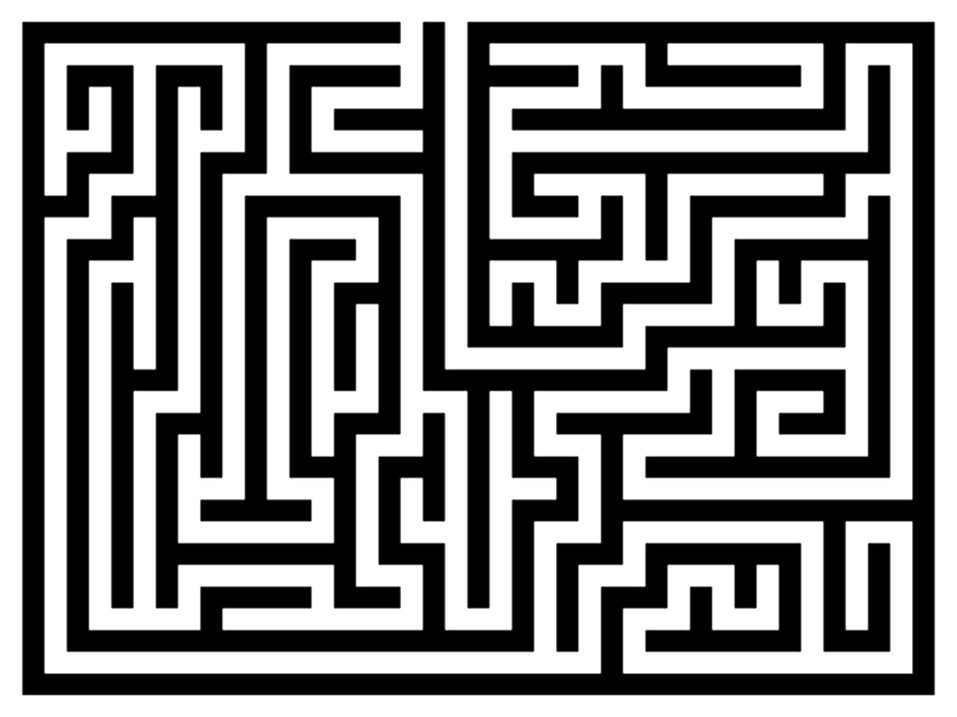


Table of Contents

1. **Introduction 3**
2. **Project Scope 5**
   1. Non-deterministic computing 5
      1. Maze Problem5
3. **Materials and Methods 19**
   1. *amb* Evaluator 5
   2. *Continuations and Continuation Passing* 5
4. **Maze Solver 20**
   1. High-level description 20
   2. Implementation details 20
5. **Further Research 24**
6. **Conclusion 30**
7. **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]](#footnote-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 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.

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. In the following sections, we will be going over our project in more details.

1. **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.

1. **Methods and Operations**
   1. ***amb* Evaluator**

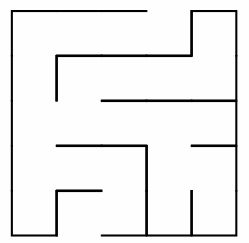
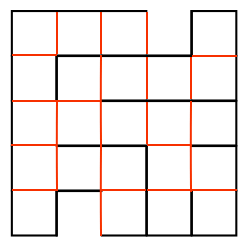
The idea of *amb* is difficult to understand, at first, but here we will try to give a high-level description of what it is and simplify things a little bit. Amb is a nondeterministic operator that takes zero or more expressions and makes an “ambiguous” (or nondeterministic) choice among them. Those 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.

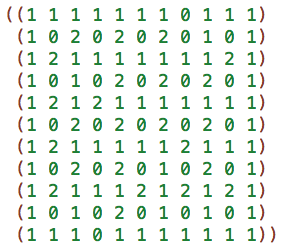
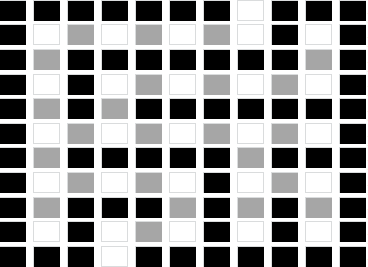
There are two types of non-determinism: *weak and strong non-determinism[[2]](#footnote-2).* The implementation of amb that we use in this project falls under the weak non-determinism umbrella[[3]](#footnote-3), where its choice follows a certain order and is not “fully” nondeterministic. In other words, 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 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.

* 1. ***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 [INSERT SOURCE HERE] 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]](#footnote-4). Here is an example in pseudo-code, assume we want to evaluate the following expression: if is *true* do , otherwise do . “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.

1. **Maze Solver** 
   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 is recursive division 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 the difference how we converted a typical maze to a Scheme maze 

In order to represent the maze shown in Figure 1.a in Scheme we had to first think of it as if it as a ”cellular grid”. The lines in Figure 1.b (both red and black) should in fact be cells if we want to represent that maze in Scheme.

There are three possible values for each cell a ‘0’, ‘1’, or ‘2’. Therefore, the resulting maze from our random-maze builder is (2n+1) (2n+1) maze, where the start-point is always at a random cell in the top-most row and the end-point is also at a random cell but in the bottom-most row. There are four directions the non-deterministic program (well call it the player) could take forward, backward, right, or left. The current position of the player gets passed to the direction’s function, which does the “moving”. The position of the player is represented in a (row, column) pair. A move to the right means that new position is (row, column+1), left is (row, column-1), forward is (row+1, column) and backward is (row-1, column). Those four directions are amb’s finite list of choices as we introduced in section 2.

Next we introduce our game rules and restrictions. Each position (row, column) 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 that new position and we store that 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 non-deterministically 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 Dead-end]

The player can neither move right, left, forward, nor backward. The later is due to our restriction a player’s new position cannot be the same as their previous positon, while the rest is because of our rule number 1, 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 that amb (the player, remember) has exhausted all options and is stuck. Furthermore, that means that our correct path list does not have the correct path to solving the matrix. That is 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 clever way of solving this problem. We allow the player to change the value of his current position if he found that this was a dead-end. Then we pop that position from the correct path list and assign their previous correct position (car of the cdr) 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 process until he does.

* 1. ***Details of implementation***

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.

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

Our solution uses a function called “iterations-start-at”. In fact, this function is a very optimistic one, where it only looks for the previous state and increments the value of the given parameter by 1. In other words, this function serves as a 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 everytime “an-element-of” function

1. **Further Research**
2. **Conclusion**

1. https://ascent.atos.net/the-rise-of-functional-programming/ [↑](#footnote-ref-1)
2. http://comjnl.oxfordjournals.org/content/35/5/514.full.pdf [↑](#footnote-ref-2)
3. we used the macro implementation of amb [↑](#footnote-ref-3)
4. http://www.scheme.com/tspl3/further.html [↑](#footnote-ref-4)