Deterministic-Finite Automaton Implementation in Functional Programming

[Hassan Almosa]

Contents

1 Demonstration 3

2 Reflection 6

3 Modification Design Strategies 7

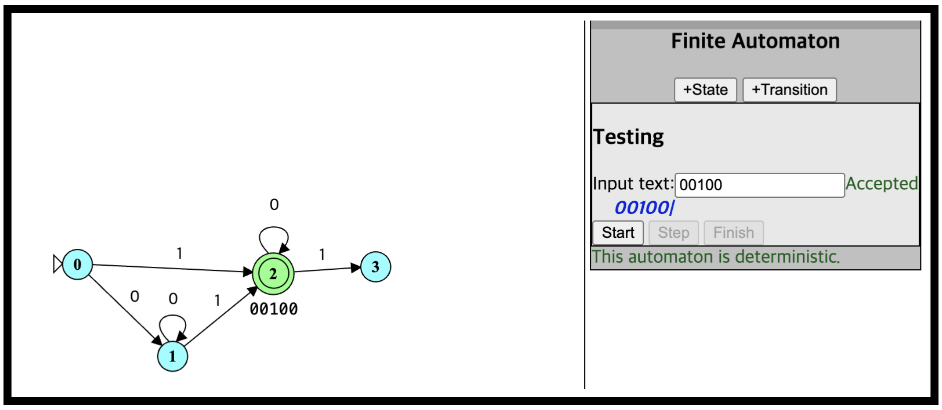
4 References 8

5 Appendix 8

Declaration of Gen AI Use

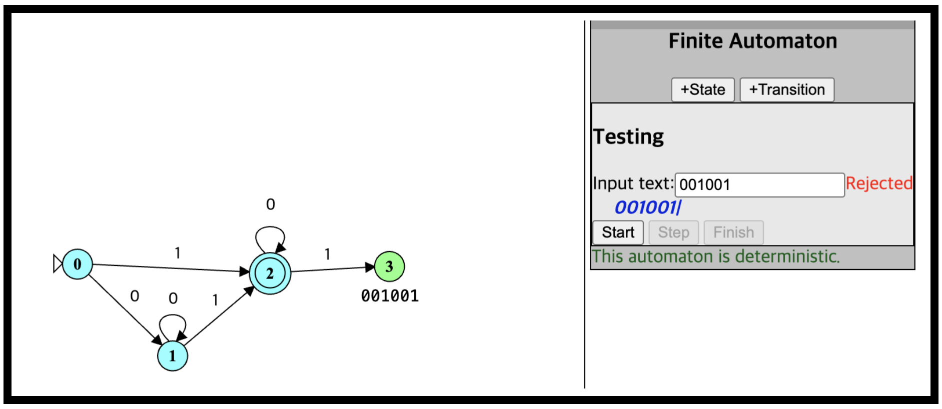
In preparing this assessment, Artificial Intelligence (AI), ChatGPT 4o/4 LLM was utilised strictly as a tool for brainstorming and structural guidance, fully adhering to the permitted uses set forth in the course policy. The AI's involvement was limited to suggesting ideas for organising content, outlining the fundamental concepts behind automata theory paradigms including Functional, Object-Oriented, and Procedural programming, and providing advice on thematic structuring for reflections and comparative analyses.
Specifically, AI assistance was sought for conceptual clarification regarding recursion, algebraic data types, and the differences between programming paradigms, as well as for recommendations on presenting the extensibility and modularity strategies relevant to automata implementations. The AI also aided in LaTex formatting, referencing, generating this statement, refining the essay structure and offered guidance on the articulation of ideas without generating any final text, code, or original analytical content.
All final submissions, including source code, written reflections, explanations, and conclusions, are the sole intellectual property and original work of the author.
Full log of AI prompts and outputs is linked in Appendix for assessor review.

1 Demonstration



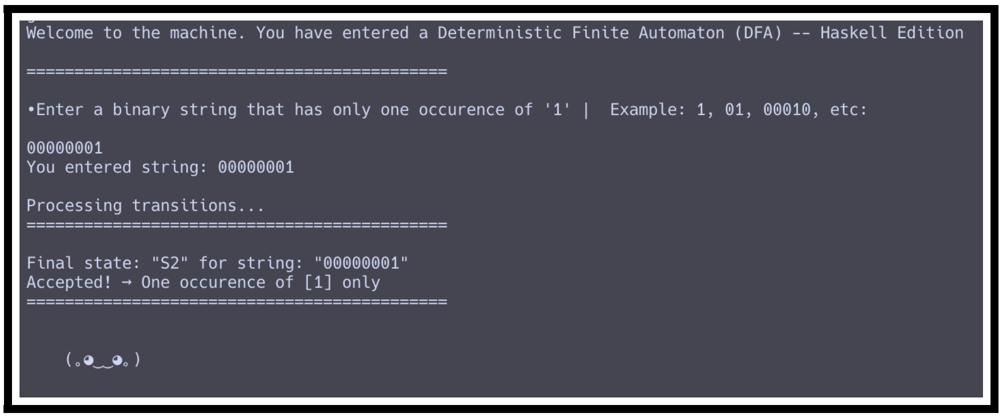
Input text: 00100 Accepted
00100|
This automaton is deterministic.

Figure 1: ACCEPT state DFA transition diagram.



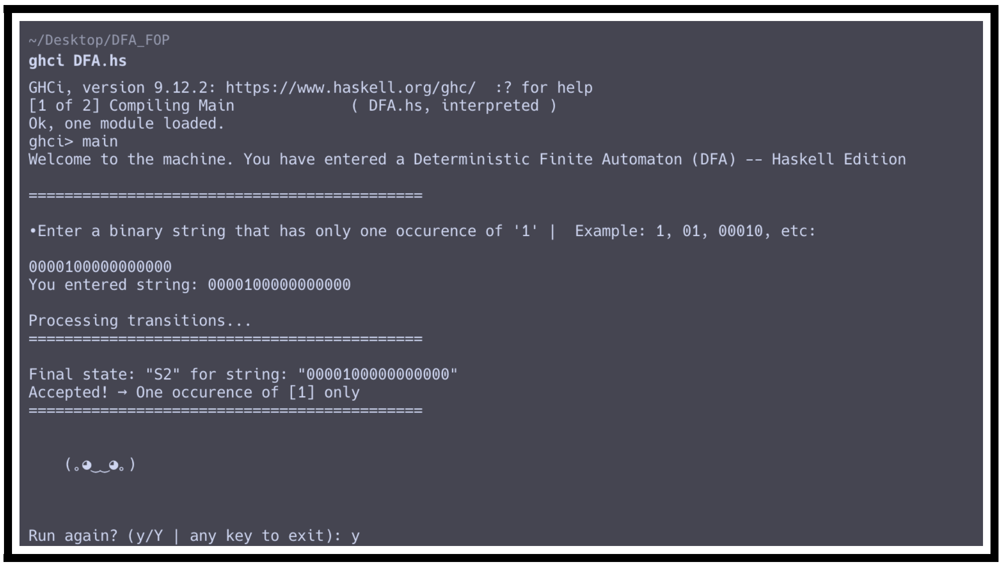
Input text: 001001 Rejected
001001|
This automaton is deterministic.

Figure 2: REJECT state DFA transition diagram.



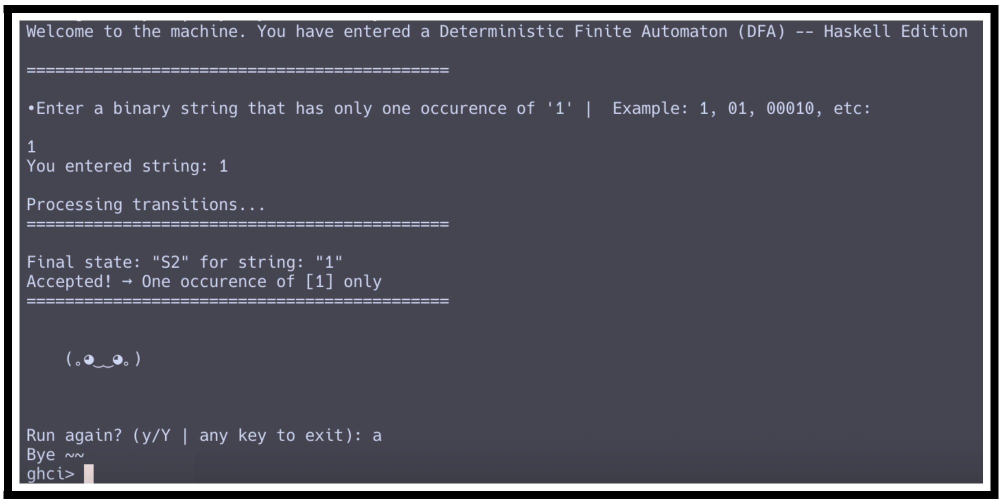
```
Welcome to the machine. You have entered a Deterministic Finite Automaton (DFA) -- Haskell Edition

============================================

•Enter a binary string that has only one occurence of '1' |  Example: 1, 01, 00010, etc:

00000001
You entered string: 00000001

Processing transitions...
============================================

Final state: "S2" for string: "00000001"
Accepted! → One occurence of [1] only
============================================


    (｡◕‿◕｡)
```

Figure 3: Example 1 of the Haskell DFA implementation correctly reaching an ACCEPT state.



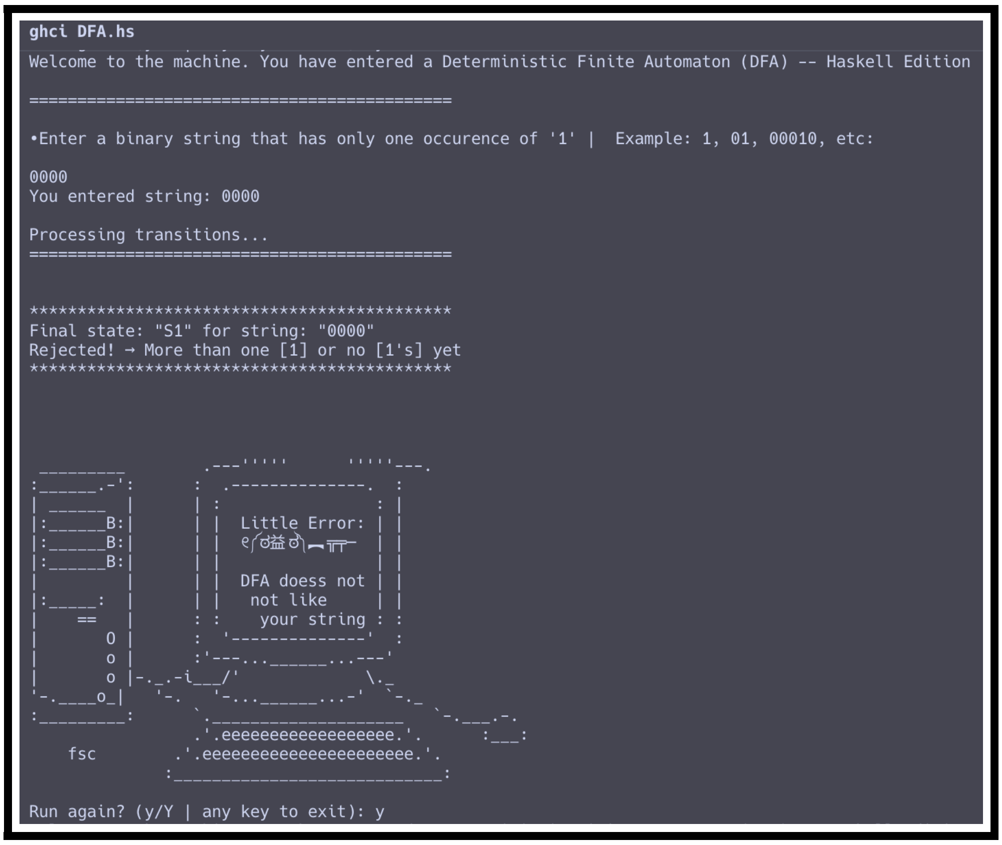
```
~/Desktop/DFA_FOP
ghci DFA.hs
GHCi, version 9.12.2: https://www.haskell.org/ghc/  :? for help
[1 of 2] Compiling Main             ( DFA.hs, interpreted )
Ok, one module loaded.
ghci> main
Welcome to the machine. You have entered a Deterministic Finite Automaton (DFA) -- Haskell Edition

============================================

•Enter a binary string that has only one occurence of '1' |  Example: 1, 01, 00010, etc:

0000100000000000
You entered string: 0000100000000000

Processing transitions...
============================================

Final state: "S2" for string: "0000100000000000"
Accepted! → One occurence of [1] only
============================================


    (｡◕‿◕｡)


Run again? (y/Y | any key to exit): y
```

Figure 4: Example 2 of the Haskell DFA implementation correctly reaching an ACCEPT state.



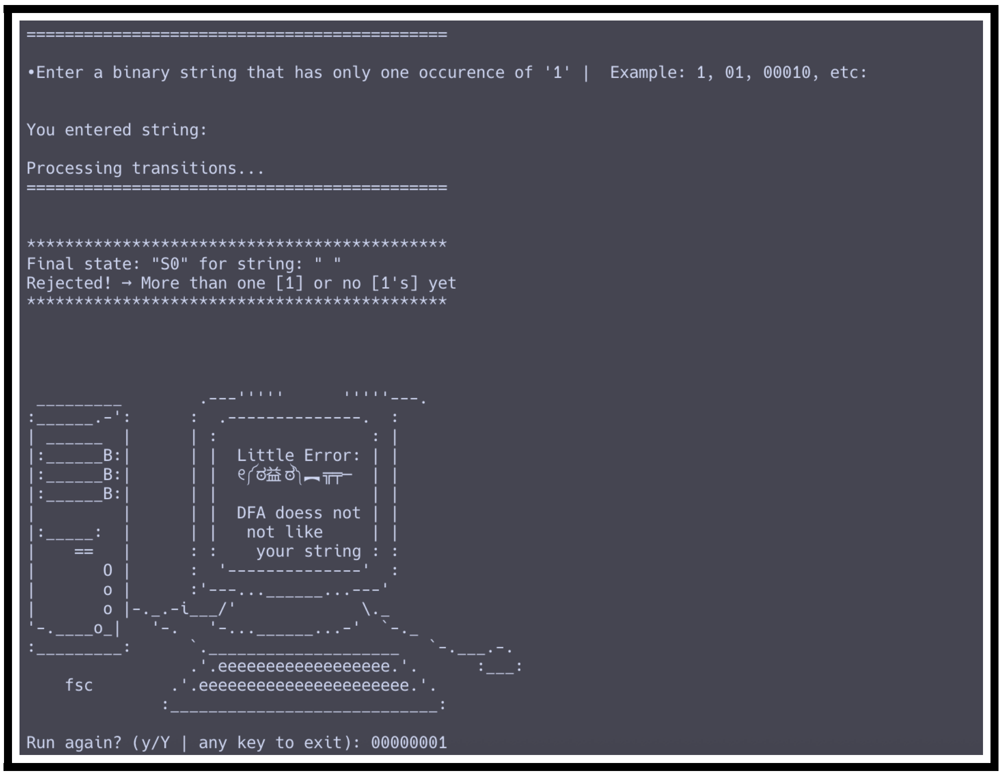
```
Welcome to the machine. You have entered a Deterministic Finite Automaton (DFA) -- Haskell Edition

============================================

•Enter a binary string that has only one occurence of '1' |  Example: 1, 01, 00010, etc:

1
You entered string: 1

Processing transitions...
============================================

Final state: "S2" for string: "1"
Accepted! → One occurence of [1] only
============================================


    (｡◕‿◕｡)


Run again? (y/Y | any key to exit): a
Bye ~~
ghci>
```

Figure 5: Example 3 of the Haskell DFA implementation correctly reaching an ACCEPT state.



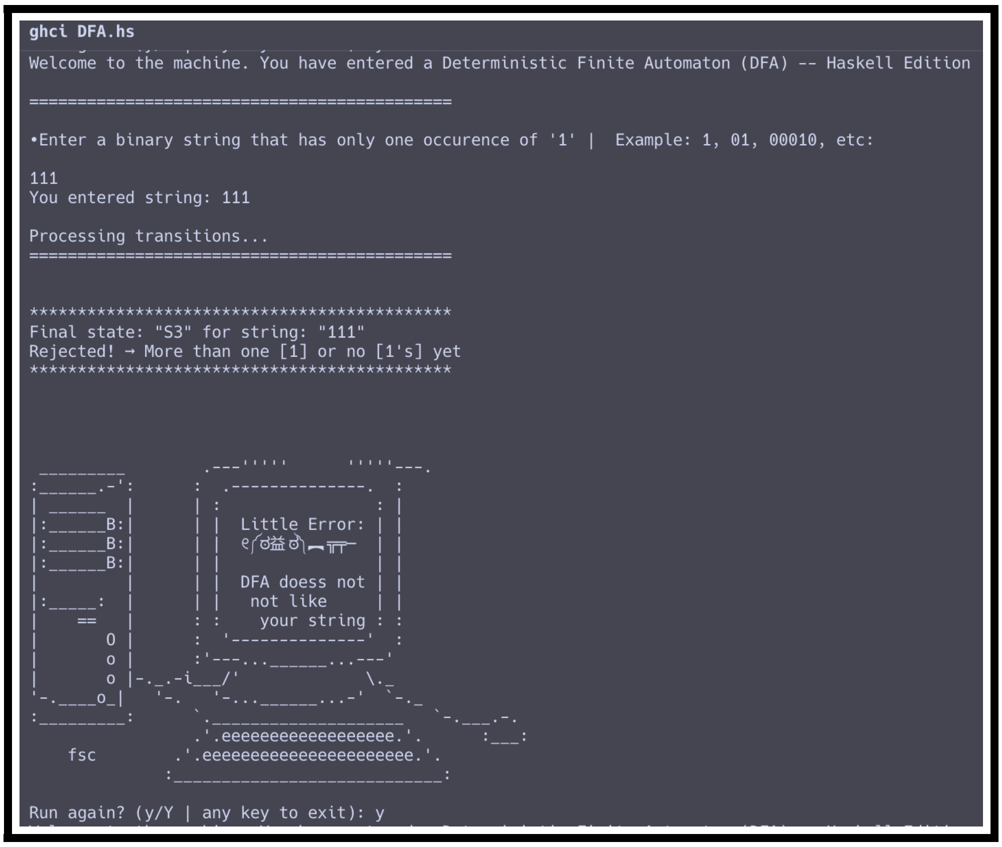
```
ghci DFA.hs
Welcome to the machine. You have entered a Deterministic Finite Automaton (DFA) -- Haskell Edition

============================================

•Enter a binary string that has only one occurence of '1' |  Example: 1, 01, 00010, etc:

0000
You entered string: 0000

Processing transitions...
============================================


********************************************
Final state: "S1" for string: "0000"
Rejected! → More than one [1] or no [1's] yet
********************************************




 _________        .---'''''      '''''---.
:______.-':      :  .--------------.  :
| ______  |      | :                : |
|:______B:|      | |  Little Error: | |
|:______B:|      | |  ୧╭ಠ益ಠ╮︻╦╤─    | |
|:______B:|      | |                | |
|         |      | |  DFA doess not | |
|:_____:  |      | |   not like     | |
|    ==   |      : :    your string : :
|       O |      :  '--------------'  :
|       o |      :'---...______...---'
|       o |-._.-i___/'             \._
'-.____o_|   '-.   '-...______...-'  `-._
:_________:        `.____________________   `-.___.-.
                  .'.eeeeeeeeeeeeeeeeee.'.      :___:
    fsc         .'.eeeeeeeeeeeeeeeeeeeeee.'.
               :____________________________:

Run again? (y/Y | any key to exit): y
```

Figure 6: Example 1 of the Haskell DFA implementation correctly reaching a REJECT state.



```
============================================

•Enter a binary string that has only one occurence of '1' |  Example: 1, 01, 00010, etc:


You entered string:

Processing transitions...
============================================


********************************************
Final state: "S0" for string: " "
Rejected! → More than one [1] or no [1's] yet
********************************************




 _________        .---'''''      '''''---.
:______.-':      :  .--------------.  :
| ______  |      | :                : |
|:______B:|      | |  Little Error: | |
|:______B:|      | |  ୧╭ಠ益ಠ╮︻╦╤─    | |
|:______B:|      | |                | |
|         |      | |  DFA doess not | |
|:_____:  |      | |   not like     | |
|    ==   |      : :    your string : :
|       O |      :  '--------------'  :
|       o |      :'---...______...---'
|       o |-._.-i___/'             \._
'-.____o_|   '-.   '-...______...-'  `-._
:_________:        `.____________________   `-.___.-.
                  .'.eeeeeeeeeeeeeeeeee.'.      :___:
    fsc         .'.eeeeeeeeeeeeeeeeeeeeee.'.
               :____________________________:

Run again? (y/Y | any key to exit): 00000001
```

Figure 7: Example 2 of the Haskell DFA implementation correctly reaching a REJECT state.



```
ghci DFA.hs
Welcome to the machine. You have entered a Deterministic Finite Automaton (DFA) -- Haskell Edition

============================================

•Enter a binary string that has only one occurence of '1' |  Example: 1, 01, 00010, etc:

111
You entered string: 111

Processing transitions...
============================================


********************************************
Final state: "S3" for string: "111"
Rejected! → More than one [1] or no [1's] yet
********************************************




 _________        .---'''''      '''''---.
:______.-':      :  .--------------.  :
| ______  |      | :                : |
|:______B:|      | |  Little Error: | |
|:______B:|      | |  ୧༼ಠ益ಠ༽︻╦╤─   | |
|:______B:|      | |                | |
|         |      | |  DFA doess not | |
|:_____:  |      | |   not like     | |
|    ==   |      : :    your string : :
|       O |      :  '--------------'  :
|       o |      :'---...______...---'
|       o |-._.-i___/'             \._
'-.____o_|   '-.   '-...______...-'  `-._
:_________:      `.____________________    `-.___.-.
                 .'.eeeeeeeeeeeeeeeeee.'.      :___:
    fsc        .'.eeeeeeeeeeeeeeeeeeeeee.'.
              :____________________________:

Run again? (y/Y | any key to exit): y
```

Figure 8: Example 3 of the Haskell DFA implementation correctly reaching a REJECT state.

2 Reflection

- **Algebraic Data Types (ADTs):** For our automaton modelling use case, ADTs are powerful structural tools for representing the `States` of the machine and treating them functionally as such, pure and immutable, in contrast with the Procedural and Object-Oriented approaches in C and C++ where they were represented as `enum` type integers. This is significantly better in terms of type safety and for ensuring that functions operating on `State` can only receive valid and defined values[1]. ADTs naturally model the abstract mathematical nature of state machines, which are fault-intolerant constructs that must be consistently reproducible[6].

- **Pattern Matching:** Implementation of `States` in ADT naturally enables pattern matching through the sum of structures. This allows for much more concise and readable declarative function definitions, replacing imperative branching as in our POP approach with generalisable functional representation of `States` through symbolic expression[1, 6].

- **Recursion:** The recursion concept in `Haskell` is a natural extension of the pure functional nature of operating on functions, replacing the mutable loop logic on counters in OOP and POP[6]. Recursive functions fit into the composable nature of functional operations, especially for our finite machines (akin to a Turing Machine of Turing Machines) processing symbolic state transitions through recursive calls.

- **Maintainability, Correctness, and Syntactical Clarity:** Haskell's strong statically typed immutable functional paradigm eliminates side effects, guaranteeing reproducible execution. The language's modular architecture, enhanced by lazy evaluation, enables highly composable components where functions are decoupled from evaluation, as code is only computed when actually needed[3]. Furthermore, Haskell's syntax naturally represents the formal DFA mathematical nature, with succinct direct syntax significantly reducing boilerplate code and logic compared to other paradigms. This reduces overhead, enhances functional focus, and improves maintainability.

3 Modification Design Strategies

Through experimenting in building the same automaton across the three programming paradigms, we gained practical insight into the strengths and weaknesses of each approach in terms of extensibility and adaptation. For the use case of scaling our DFA to recognise new language specifications, the strategies differ fundamentally from imperative paradigms. While OOP provides extensibility mechanisms with associated overhead, FOP avoids this by treating logic as data, which introduces its own trade-offs at larger scales.

In our **C POP implementation**, the automaton's logic is static and hard-coded. The `States` and the `transition` function are compiled directly into the executable. To adapt the DFA to a new language would require direct modification of the `transition` function for each change and its dependencies; it is simple yet brittle, side-effect prone, and difficult to scale.

The **OOP implementation** offered a significant improvement in structure and extensibility through polymorphism[4]. By defining an abstract `DasAutomat` base class, and deriving from this class to create child classes encapsulating specific automaton logic[1], we achieved flexibility suitable for extending automata to different languages.

In our **FOP Haskell implementation**, we saw a more elegant and powerful approach for this specific automaton, shifting from an imperative to a declarative model, opening up better strategies. These include:

- **Parameterisation via Higher-Order Functions:** The `runDFA` function can be made generic by accepting the transition logic and final-state function as arguments, decoupling logic from specification[1].
- **Data-Driven Design:** A more advanced strategy involves modelling the entire DFA specification as a pure, immutable data structure[1]. This allows a universal `runDFA` function to interpret any DFA passed to it. This also supports DSLs (domain-specific languages) for automata declaration in mathematical notation that compiles directly to specifications, enhancing expressiveness and type safety.
- **Abstraction with Typeclasses:** Haskell's `typeclass` system allows us to define shared interfaces for automata without inheritance[1], equivalent to interfaces in Java or C++.
- **State Management with Monads:** For complex machines, immutable state requires passing explicit state through transitions. This is handled via the `State` monad, encapsulating state logic without mutation[1].
- **Superior Handling of Non-Determinism:** Although was not a part of our automata implementation, its notably worthy to point where functional programming truly excels at. In modelling non-deterministic automata like NPDAs, Haskell's `list monad` and lazy

evaluation naturally handle non-deterministic branching computation paths naturally and perfectly well. ADTs ensure thread-safe, concurrent state explorations without synchronisation mechanisms.

Concluding, by looking at all three implementations, the functional paradigm offers the most mathematically elegant solution. The automaton truly becomes what it conceptually is, a pure function from inputs to states, no hidden mutations, no complex object hierarchies and human centred constructs, much better befitting software to represent natural phenomena and universal laws in a sense. Each paradigm has its place, but for automata theory its the most intuitive approach thus far.

4 References

[1] Gen AI LLM Assistant (Explanations, Coding Questions, Idea Generation and Planning, Code Comments, Debugging, Report Planning, Document Formatting). OpenAI, ChatGPT 4o/4.

[2] Daily Dev. "Haskell for Beginners." daily.dev, https://daily.dev/blog/haskell-for-beginners. Accessed June 2025.

[3] Tadele, M. "Haskell Programming Language: Introduction." Medium, https://medium.com/@mearaftadewos/haskell-programming-language-introduction-dac99443a10d. Accessed June 2025.

[4] Reddit – r/SoftwareEngineering. "What Are the Pros and Cons of Using Functional Programming?" https://www.reddit.com/r/SoftwareEngineering/comments/jpsmhf/what_are_the_pros_and_cons_of_using_functional/. Accessed June 2025.

[5] Reddit – r/cscareerquestions. "OOP vs Functional Programming." https://www.reddit.com/r/cscareerquestions/comments/10es6lu/oop_vs_functional_programming/. Accessed June 2025.

[6] Scaler Topics. "Java: OOP vs Functional vs Procedural Programming." https://www.scaler.com/topics/java/oop-vs-functional-vs-procedural/. Accessed June 2025.

5 Appendix

- **ChatGPT Conversation Log:** https://chatgpt.com/share/685c1c39-2574-8011-8f4f-af7a9bf83a67