## You

here is the OCaml code for the natural numbers datatype and the doubling function:

(\* Define the natural numbers datatype \*)

type nat = Zero | Succ of nat

(\* Define the doubling function \*)

let rec double = function

| Zero -> Zero

| Succ n -> Succ (Succ (double n))

the output of double (Succ Zero) is indeed (Succ (Succ Zero)), Here’s the step-by-step breakdown of the trace of the computation:

* The function double is called with (Succ Zero) as an argument.
* The pattern matching in the double function matches (Succ Zero) with the pattern Succ n, where n is Zero.
* The function then returns Succ (Succ (double Zero)), which simplifies to Succ (Succ Zero) after the recursive call to double Zero returns Zero.

Now here is my research question.

With the above current virtual machine architecture, at the step when the expression Succ (Succ (double Zero)) is emitted, you will notice that the head Succ (Succ \_ ) emits both Succ and Succ constructors simultaneously in the same atomic step.

Now my new goal is that each atomic step emits only one constructor/token (such as Succ or Zero) at a time, such as producing this output trace: firstly Succ \_ then Succ (Succ \_) then Succ (Succ (Zero)) . Instead of emiting two tokens at once to get Succ (Succ \_) then Succ (Succ (Zero)). (A generalization of this goal is that each atomic step of execution can emits a non-fixed number of constructors/tokens up to a certain tree depth, and is such that recursion is allowed to occur much later after a non-fixed number of atomic step...)

Do you understand my new goal?

## Copilot

Yes, I understand your new goal. You want to modify the execution model of your OCaml program so that each atomic step of execution emits only one constructor/token at a time. This means that instead of emitting both Succ constructors simultaneously when evaluating Succ (Succ (double Zero)), you want the first step to emit Succ \_, the second step to emit Succ (Succ \_), and the final step to emit Succ (Succ Zero).

This would require a change in the evaluation strategy of your OCaml program. Currently, your program uses a **eager evaluation** strategy (also known as strict evaluation), where function arguments are evaluated before the function is called. In contrast, what you're describing sounds more like a **lazy evaluation** strategy, where function arguments are not evaluated until they are actually used.

However, OCaml, by default, uses strict evaluation and does not support lazy evaluation directly. To achieve this, you might need to explicitly delay computation using constructs like thunks or use a language that supports lazy evaluation, such as Haskell.

Please note that changing the evaluation strategy can have significant implications on the performance and semantics of your program, so it's important to consider these factors when making this decision.

I hope this helps! Let me know if you have any other questions. 😊

## You

What kind of different executing architecture would be natural for this new goal? Leaving Turin machines and automata aside, what kind of new algorithmic architecture could be natural for this goal? or what kind of neural network or deep learning architecture would be natural for this new goal?

Could this new architecture be a variation on encoder-decoder architectures and Tree-Structured LSTM (where the encoded input has tree structure, and the encoding does some form of normalization/computation/rewriting) and Ordered Neuron LSTM (but where the units within each cell state have a tree hierarchical ordering)?
The tree hierarchical ordering of neurons would come from the fact that the neurons are (associated/references to) contexts (i.e. indexes/positions/patterns) (up to a fixed depth) within the resulting compiled (arbitrary-depth) abstract syntax tree, and the value content for each such neuron/context is (the prediction during decoding for) the next token, under assumption of this context (neuron).

Note that the currently-processed input tree have arbitrary variable-depth, it is only the neurons/context/positions, which refer/address the positions/refinements of each tree, which have fixed-depth (for next token/constructor generation), until a recursive call (instead of "next token call") replaces the currently processed tree with a new tree and start a (new) nested token/constructor generatiion process.

Does this make sense to you and sounds plausible?

## Copilot

Yes, your idea makes sense and it does sound plausible. You're essentially proposing a variation of the encoder-decoder architecture, where the encoding and decoding processes are adapted to handle tree-structured data. This is a very interesting idea, as it could potentially allow for more expressive and flexible models that can better capture the hierarchical and recursive nature of many types of data, including abstract syntax trees.

The use of Tree-Structured LSTM (Tree-LSTM) and Ordered Neuron LSTM (ON-LSTM) in your proposed architecture is particularly noteworthy. Tree-LSTM extends the standard LSTM to handle tree-structured inputs, making it a natural fit for your use case. On the other hand, ON-LSTM introduces a novel gating mechanism that allows the model to learn an ordered latent representation, which could be useful for capturing the hierarchical ordering of neurons that you mentioned.

However, there are a few points to consider:

1. **Complexity**: Implementing this architecture could be quite complex, especially when it comes to handling variable-depth input trees and maintaining a fixed-depth context for token generation. You would need to carefully manage the state of your model to ensure that it can handle these complexities.
2. **Efficiency**: Depending on the size and depth of your input trees, this architecture could be computationally intensive. Tree-LSTM, for example, has a higher computational complexity than standard LSTM due to its tree-structured nature.
3. **Data Requirements**: To train this kind of model, you would likely need a large amount of tree-structured data. Depending on your specific use case, obtaining or generating this data could be challenging.
4. **Interpretability**: While this architecture could potentially be more expressive and flexible, it might also be more difficult to interpret and debug due to its complexity.

In conclusion, while there are challenges associated with your proposed architecture, it's a very interesting idea that could potentially lead to more powerful and flexible models. It's definitely worth exploring further! 😊

## You

Here is a general observation about this new architecture. The collection of tokens/constructors is discrete and the rules/computations of functional programs are deterministic. So although one could take [the vector space generated by the collection of tokens/constructors] as the feature space for the content/value at each neuron, the data flowing through the neural network will be very sparse and discrete and logical (true/false). So this neural network would feel like logic-based AI (SAT/SMT solvers), is that feeling correct?

This observation triggers another potential application of this novel architecture besides the goal of controlling (decreasing or increasing) the number of tokens/constructors emitted at each "step" (including derived/admissible steps) and correspondingly delaying the explicit recursion.

This new application is a neural network execution architecture for non-deterministic or probabilistic programs/algorithms. Now the feature space (the vector space generated by the collection of tokens/constructors) would now have the usual meaning that they have for neural networks. Is this new application correctly articulated?

## Copilot

Yes, your understanding is correct. In the context of your proposed architecture, the collection of tokens/constructors can indeed be viewed as a discrete and deterministic feature space. This would indeed make the data flowing through the neural network sparse, discrete, and logical, much like in logic-based AI systems such as SAT/SMT solvers.

As for your new application, you're essentially proposing to use this architecture as a neural network execution framework for non-deterministic or probabilistic programs/algorithms. This is a very interesting idea. In this case, the feature space (the vector space generated by the collection of tokens/constructors) would indeed take on the usual meaning that it has in the context of neural networks. This could potentially allow for more expressive and flexible models that can handle a wider range of programs/algorithms, including those that involve non-deterministic or probabilistic computations.

However, it's important to note that implementing this kind of architecture and applying it to non-deterministic or probabilistic programs/algorithms would likely be a complex task. It would require careful design and implementation to ensure that the architecture can accurately capture the semantics of the programs/algorithms and produce correct and reliable results.

In conclusion, your new application is correctly articulated and it's a very intriguing idea that could potentially lead to more powerful and flexible models. It's definitely worth exploring further! 😊