TreeLang : a Language for Describing and Executing Rollup Tree

Haozhe Chen (Language Guru),
Yihan Shen (System Architect), Andrew Yang (Manager)
Xintong Yu (Tester)

Februrary 2023

1 Motivation

TreeLang is a novel programming language designed to facilitate tree-structured computations that include both forward and backward propagation. Many real-world problems, particularly those involving large language models (LLMs) and knowledge retrieval, can be naturally modeled as a tree structure, with forward computations producing child nodes from parent nodes, and backward computations aggregating information from children to parents. TreeLang provides a framework for easily defining and manipulating such tree structures, allowing users to describe, edit, and extend existing trees, as well as define custom forward and backward operations.

The primary motivation for TreeLang stems from the need for more sophisticated knowledge retrieval and reasoning capabilities in LLMs, particularly in domains where an LLM might need to iteratively retrieve and integrate information from multiple sources to arrive at a comprehensive picture. In healthcare, for example, a reasonable diagnosis or treatment recommendation demands many rounds of information retrieval and evaluation. Existing retrieval-augmented systems typically retrieve only a single document or observation based on the initial input, limiting their ability to perform multi-step reasoning and knowledge aggregation. TreeLang aims to bridge this gap by enabling complex, tree-structured retrieval processes, where each node in the tree represents a stage of observation, reflection, or intention, and the forward and backward passes facilitate the retrieval and aggregation of relevant information. By drawing inspiration from works like Tree-of-thoughts[2] and entailment trees[1], TreeLang provides a powerful framework for enhancing the reasoning capabilities of LLMs and enabling more intricate, context-aware knowledge retrieval and integration.

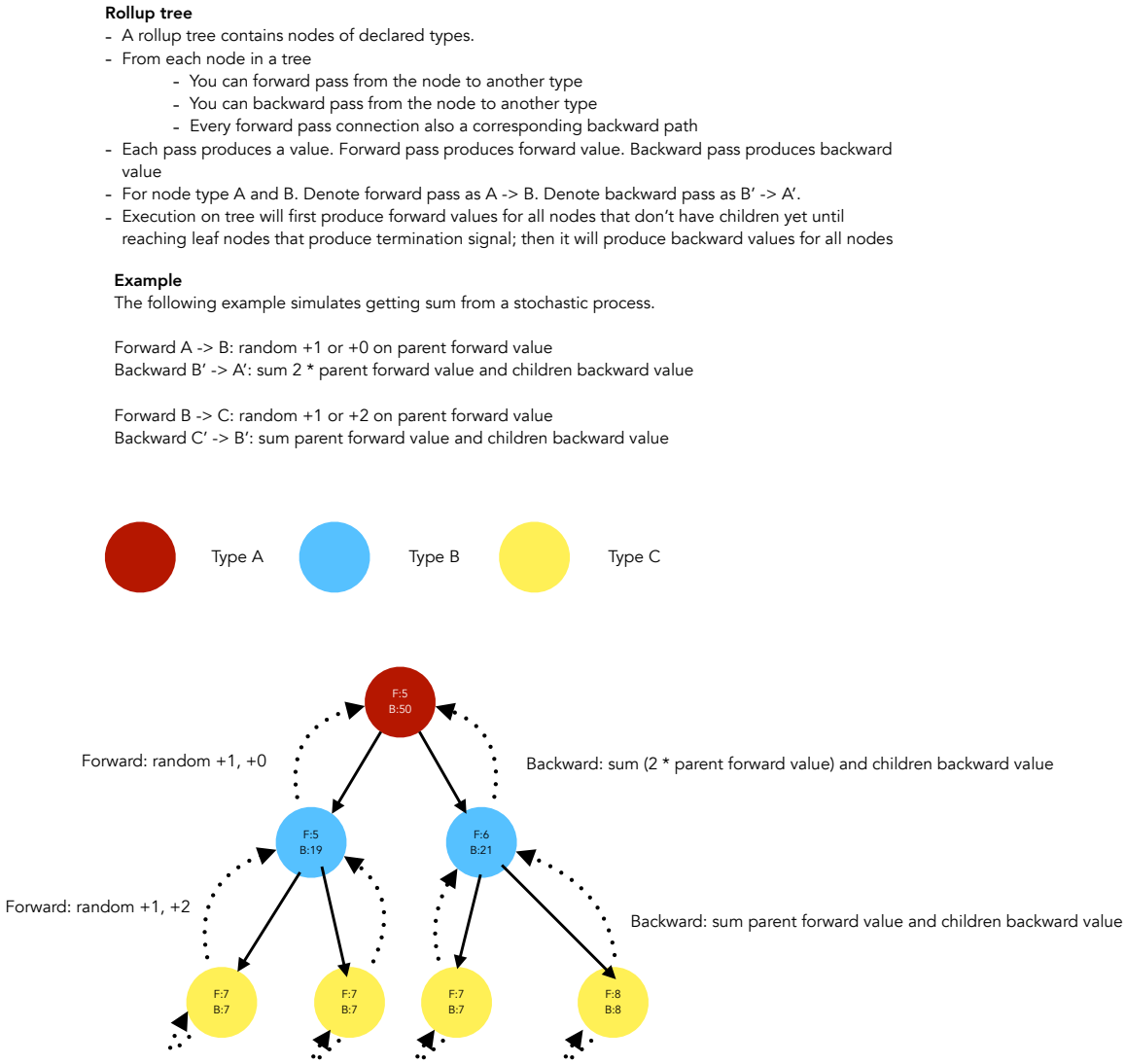


Figure 1: Explanation and example of rollup tree. Forward values are denoted with F, and backward values are denoted with B.

2 Language Tutorial

We propose `TreeLang` to deal with tree-structured computations that requires forward computation and backward computation. We call this structure a rollup tree. Forward computations produce children from parents. Backward computations aggregate children information to parents. We show description of what a rollout tree is in Figure 1. We also show an example tree computation.

`TreeLang` allows an user to

- Describe an existing tree (in lazy fashion; i.e. user can describe different parts of tree

in no particular order)

- Edit the existing tree
- Define forward and backward operations
- Run succeeding operations to expand the tree

`TreeLang` includes two components.

The first component `TML` (Tree Markup Language) is a **declarative**, strong-static typed, dynamic scope, lazy evaluation language that allows description of an existing tree in a succinct, intuitive and simple way (kind of analogous to HTML in front-end development).

The second component `TScript` (Tree Script) is a **imperative**, strongly-static typed, static scope, eager evaluation language that takes actions with or without knowledge to the tree as is defined in the associated .tml file. It may act on the tree as defined in the .tml to do certain analysis, add or modify information on the tree, define and execute forward and backward passes on existing trees. Notice that the key difference of our language is that the tree as is defined in the .tml is just the starting point, and .tscript allows users to define a series of actions to keep augmenting the tree.

Why dynamic scoping? It is important to notice that we free the users from having complete knowledge of the graph and carefully plan out how the graph should look like. Instead, user can define any part of the graph in any order they feel like: for example, the user may say node B is connected to A as a child and has a value of 5 before the node A was even defined! Eventually, it is up to us to check if the tree makes sense and can be compiled into a tree; and, if it does compile, then visualize the tree for the user.

Why strong typing? We want to ensure the safety and predictability of strong typing. This approach is suitable for a language designed to handle complex tree structures and operations.

Why strict evaluation order? Our programming language incorporates two files: one for graph definition and the other for execution, which includes forward computation and backward propagation. In the former, we declare abstract node type and construct a tree structure without specifying the exact data flow(i,e, actual node values/how nodes would pass along their values). This file uses lazy evaluation. In the latter file, as the users input the exact functions that would act on each node, we deploy a strict evaluation order.

3 Architectural Design

In the section above we have demonstrated the basic syntax for both the tree markup file (.trml) and tree script file (.trs). The former demonstrates instancing (forward) connection of nodes while the latter shows code that will create a single Leaf node for the Root with the same value as its parent. Now we will give a short introduction to each of the components involved in the translator and the interfaces that we implemnted.

- Scanner (scanner.ml): The scanner takes the input code as a stream of characters and produces a stream of tokens. It interfaces with the parser by providing the next token when requested. Our scanner acts like generic scanner except that we have created a few special tokens such as CONNECT, ROOT, and FORWARD/BACKWARD PREPARATIONS.

- Parser (treelangparse.mly): The parser takes the stream of tokens from the scanner and constructs an Abstract Syntax Tree (AST) representation of the program. It interfaces with the AST module by creating and returning AST nodes based on the language's grammar rules. In our specific implementation, we allow four types of declarations: node type declarations, connect declarations, node creation declarations, and operation declarations, which support the most fundamental usage of our language.

- AST (ast.ml): The AST module defines the data structures and types used to represent the program's syntax tree. It provides functions for creating, manipulating, and pretty-printing the AST nodes. It interfaces with the Semant module by providing the AST representation of the program.

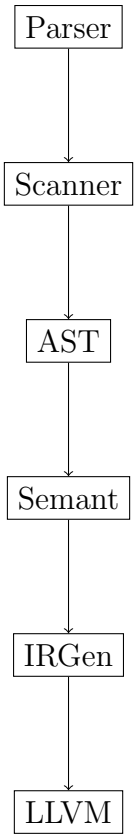
- Semant (semant.ml): The Semant module performs semantic analysis on the AST, checking for type errors and other semantic issues. It produces a Semantically Checked Abstract Syntax Tree (SAST) if the program is semantically correct. It interfaces with the IRGen module by providing the SAST representation of the program.

- IRGen (irgen.ml): The IRGen module takes the SAST and generates LLVM Intermediate Representation (IR) code for the program. This part is still work in progress.

- LLVM: The LLVM component takes the generated LLVM IR code and performs further optimization, code generation, and executable linking. This file is also a work in progress.

- Treelang (treelang.ml): The main driver program that coordinates the compilation process. It handles command-line arguments, reads input files, and calls the appropriate components based on the specified actions (printing AST, SAST, or LLVM IR).

Here's a schematic diagram of the different parts of our translator.



Yihan Shen and Andrew Yang was mainly responsible for implementing the scanner and parser. Haozhe Chen was mainly responsible for implementing the semantics checker and IR generation. Xintong Yu was responsible for writing test cases and debugging code in each phase of the compilation.

4 Test Plan

4.1 Representative Tests

- 1. (Scanner) Test Complex Identifiers

  Source Code:

  node123 DirectoryNode int

  Expected Tokens: [ID("node123"), ID("DirectoryNode"), INT_TYPE]

  Purpose: Validates that the scanner correctly identifies complex alphanumeric identifiers and keywords.

- 2. (Parser)Test Nested Structures

  Source Code:

  node X int int

node Y int int

x → X x1 = 50

x1 → Y y1 = 150

Expected AST Output:

[NodeTypeDecl(Node, "X", IntType, IntType),

NodeTypeDecl(Node, "Y", IntType, IntType),

NodeCreationDecl("x", "X", "x1", Some(IntLit(50))),

NodeCreationDecl("x1", "Y", "y1", Some(IntLit(150)))]

Purpose: Tests the parser's ability to handle complex structures involving multiple declarations and connections.

- 3 (IR Generation) Test multiple Connected Nodes.

  Source Code:

  Root root int int

  node Directory int int

  node File int int

  root → Directory dir1 = 200

  dir1 → File file1 = 100

  Generated LLVM IR:

  ; ModuleID = 'TreeStats'

  define i32 @main()

  entry: %num_nodes_str = global_stringptr("Number of nodes: 3")

  %max_depth_str = global_stringptr("Max depth: 2")

  call i32 (i8*, ...) printf(i8* %num_nodes_str)

  call i32 (i8*, ...) printf(i8* %max$_d$*epth_str*)

  ret i32 0

  Statistics Output

  Number of nodes: 3

  Max depth: 2

  Purpose Tests the compiler's ability to handle complex tree structures and compute correct IR code and statistics.

4.2 Automation

In testing the TreeLang compiler, we employed a robust automation strategy leveraging OCaml's testing frameworks like OUnit2, alongside tools such as Ocamllex and OCamlyacc for lexical and syntactic analysis respectively. Automated tests were systematically designed for each component—scanner, parser, semantic analysis and ir generation—to ensure accurate token generation, AST construction, semantic validation, and IR code.

For each phase of compilation, we wrote a helper function to call the corresponding compilation code to execute the input and obtain the output. The testing suite was set up to automatically execute a series of predefined test cases, comparing the generated output against expected outcomes for both correctness and error handling.

5 Summary

5.1 Work Distribution

- Yihan Shen and Andrew Yang: Implementation of the Scanner, Parser and the general design of the language.
- Haozhe Chen: Implementation of the Semantics checker and IR generation code.
- Xintong Yu: Design and implementation of test cases, and general debugging.

5.2 Important Takeaways

- Yihan Shen: My biggest takeaway is from the implementation of the scanner and the parser. I find it super interesting to see the algorithms on the slides being implemented in Ocaml code.
- Andrew Yang the the: My most important takeaway from this project is team management. Since everyone has different deadlines and different priorities, it is difficult to make sure everyone is on the same page at each stage of our implementation. However, good communication helps us a lot in finishing the project during the challenging semester.
- Haozhe Chen: My most important takeaway is the IR generation implementation. I find it interesting to see out code getting translated into executable code.
- Xintong Yu: My most important takeaway from this project is to always test new and existing code. Catching bugs and ensuring our language holds up under different scenarios was both challenging and rewarding

5.3 Advice for Future Teams

Start the project early!

References

[1] Ribeiro, D., Wang, S., Ma, X., Dong, R., Wei, X., Zhu, H., Chen, X., Huang, Z., Xu, P., Arnold, A., and Roth, D. Entailment tree explanations via iterative retrieval-generation reasoner, 2022.

[2] Yao, S., Yu, D., Zhao, J., Shafran, I., Griffiths, T. L., Cao, Y., and Narasimhan, K. Tree of thoughts: Deliberate problem solving with large language models, 2023.