Eötvös Loránd University

Faculty of Informatics

Department of Software Technology

Program Synthesis With a Stack-Based Concatenative Language

Supervisor:

Dr. Pintér Balázs

Senior Lecturer

Author:

Su Xiaotian

Computer Science BSc

Budapest, 2021

EÖTVÖS LORÁND UNIVERSITY

FACULTY OF INFORMATICS

Thesis Registration Form

Student's Data:

Student's Name: Su Xiaotian

Student's Neptun code: BS8TLS

Course Data:

Student's Major: Computer Science BSc

I have an internal supervisor

Internal Supervisor's Name: *Pinter Balazs*

Supervisor's Home Institution: *Eötvös Loránd University*

Address of Supervisor's Home Institution: *Budapest, Egyetem tér 1-3, 1053*

Supervisor's Position and Degree: *senior lecturer, PhD*

Thesis Title: Program synthesis with a stack-based concatenative language

Topic of the Thesis:

(Upon consulting with your supervisor, give a 150-300-word-long synopsis os your planned thesis.)

This project aims to develop a program synthesis model to generate automatic code given sample input-output pairs. In the model, I will train a neural network to predict a composition of terms that transform the given inputs to the given outputs. The inputs and outputs are stored in stacks, the program will be able to process them using tree-lstm and with the help of the state-of-the-art best-first beam search, it can generate a concatenative program.

The DSL used in this project is a stack-based concatenative language inspired by the Cat Programming Language which does not have variables, only a list of operations where every operation manipulates a global stack. Each operation is either a command or a value. All commands in this DSL are functions that take a stack as input and return a new stack as output. Quoted is a special command in support of higher-order functions: it represents an instruction as a value and pushes it onto the stack which can be used by other commands later. For example: <6 7 dup mul sub> results in a stack with the value 43 on top.

Budapest, 2020.11.26.

Contents

Chapter 1

Introduction

Program synthesis is the task of automatically generating code in a particular programming language that satisfies the user intent expressed in the form of some specification.

This project develops a program synthesis model to automatically construct a program given sample input and output pairs. The program is written in a Domain Specific Language that we constructed on our own.

We first designed and implemented the DSL. Then generating the random stacks and programs. Restrictions to types and values were added during data generation. Afterwards, we embedded the synthetic data into an abstract syntax tree so that they could be fed into the Tree-LSTM network. We trained the model to one term each time that transforms the given inputs to the given outputs.

The work employs beam search to the program synthesis problem. Given the current state, our neural network directly predicts the next term to manipulate the global stack to get closer to the solution. The beam search algorithm will then rank and select from the network's outputs, reapplying the predictions at each step. Eventually, we can generate a concatenative program to achieve certain functionalities.

For demonstration purposes, we added a handy graphical user interface with the PySimpleGUI library. The user can enter input and output examples and get the program that satisfies the specification. There is also an animation to exhibit how our DSL manipulates the global stack so as to get the output from the input.

The chapters are as follows: Background (chapter 1) covers the theoretical background practical necessity for program synthesis, the User Documentation(chapter 2) provides detailed guidance for the user to build dependencies and run the program, and the Developer Documentation (chapter 3) covers the necessary knowledge for implementing the domain-specific language, building and training the model, and testing.

1.1 Background

Program synthesis is a field at the intersection of programming languages, formal methods, and AI. It is the mechanized construction of software, dubbed 'self-writing code'. Synthesis tools relieve the programmer from thinking about how the problem is to be solved; instead, the programmer only provides a description of what is to be achieved. Given a specification of what the program should do, the synthesizer generates an implementation that provably satisfies this specification. Since the inception of artificial intelligence in the 1950s, this problem has been considered the holy grail of Computer Science.[15]

In the thesis, we study program synthesis in a Domain Specific Language (DSL) - a stack-based concatenative language that we implemented on our own, mostly inspired by Cat programming language [10].

Programming languages can be categorized in different ways. One of them is to define the languages as either "concatenative" or "applicative". [3] In an applicative language, things are evaluated by applying functions to arguments, like C, Python, and Java. In a concatenative programming language, things are evaluated by function compositions. There are no variables in this language, a sequence of operations will take values from an implicit data structure to operate on, and return the result to that structure. This implicit data structure is usually stack. Most existing concatenative languages are stack-based [4]. The attributes of the concatenative language make it perfect for chaining existing code to create something new.

A stack-based language is one where most operations are done on a stack of values. Functions take their inputs from the stack, do some computation, and return their results on the same stack. The language often provide some sort of stack manipulation operators. Commonly provided are `dup`, to duplicate the element atop the stack, `exch` (or `swap`), to interchange elements atop the stack. Stack-oriented programming languages may be conceptually difficult for humans to understand, but the advantage is that they are very easy for computers to evaluate and generate. Therefore, it serves as an ideal candidate for the program synthesis project. Section 2.3.1 provides a detailed explanation on the stack-based language and Section 2.3.3 digs into our DSL.

1.2 Related Work

The influential work in the field of program synthesis is DeepCoder [1] which serves as a baseline for a lot of program synthesis projects. For example, using the Domain Specific Language (DSL) defined by the DeepCoder paper, PCCoder [29] showed great advances in the performance of the synthesis process. They managed to solve programs more than two times as complex while preserving decent accuracy and search time. This software used a similar idea as shown in [1], but we constructed our own domain-specific language which is both powerful and neat.

1.3 Outline

Chapter 2 User Documentation has the following content:

- A brief introduction to the main methods and tools used to build the software. See section 2.1.
- The environment of the developer and how the end-user can install and run the program. See section 2.2.
- The description of the domain-specific language used in the software. See section 2.3.4.
- A description of system functions, the running outcome from the user's point of view, and explanation of error messages. 2.4

Chapter 3 Developer Documentation provides the following information:

- An overview of the file structure and workflow of our project. See section 3.1.
- The importance of domain-specific language and the specification and a detailed explanation of our DSL. See section 2.3.
- The process of preparing data before training the model. See section 3.3.
- The architecture of our neural network and the selection of hyperparameters and the loss function. See section 3.4.
- Testing plans, tools, writing examples, and results. Pytest [18] is for unit testing and hypothesis test [20] is used for property testing. See section 3.5.

Chapter 2

User Documentation

This software intends to demonstrate the user how our domain specific language works to manipulate the global stack. In this chapter, the application will be explained from an end-user perspective with demonstration on how to build and run the program and how to enter input-output pairs to get the program that achieve a certain function.

2.1 Main Methods and Tools

The user interface is written using PySimpleGUI [22], a Python package that enables Python programmers of all levels to create GUIs. PySimpleGUI is currently capable of running on 4 Python GUI Frameworks, they are PySimpleGUI, PySimpleGUIQt, PySimpleGUIWx, and PySimpleGUIWeb. Here we used the default PySimpleGUI Framework. The algorithm that implements the DSL and builds the model is written in Python 3. We used the typing module which was introduced in Python 3.5 [14] to do the type check and the package PyTorch [24] to build the neural nets.

2.2 Installing

The installation of the software is relatively straightforward and simple. This section will walk the end-user through the process of setting up the environment, installing dependencies, and running the program.

2.2.1 Environment

The development environment is typically a workstation for developers, while the production environment is a real-time setting where programs are run and used by end users. In this project, we will implement the app and run the tests in the development environment.

The specification of the system used to develop this application consists of:

- Operating System: macOS Big Sur
- Ram: 16.0 GB
- CPU: 2.0 GHz
- Code Editor: Visual Studio Code
- Python 3.8.0 (requirement: Python 3.5, 3.6, 3.7, 3.8)

2.2.2 Installation

We have all the prerequisites written in requirements.txt and environment.yml, you can built dependencies easily with those files in either of the following methods. Method 1: If you have `Python`, `pip` installed, you can use the following steps:

1. Open a terminal and change to the project directory.
    - Install virtualenv: `pip install virtualenv`
    - Create a virtual environment: `virtualenv venv`
    - Activate it: `source venv/bin/activate`
2. Install the requirements: `pip install -r requirements.txt`
3. Run the app: `python main.py`

Method 2: If you have `conda` installed, you can follow the steps below to install the dependencies before running the program:

1. Open a terminal and change to the project directory.
    - Create an environment: `conda env create -f environment.yml`
    - Activate the environment: `conda activate stack-prog-synth`
2. Run the app: `python main.py`

2.3 Domain Specific Language

A domain-specific language (DSL) is a small, usually declarative, language that offers expressive power focused on a particular problem domain [9]. For the task of program synthesis, the choice of DSL is important. It should be expressive enough to capture the problems that we wish to solve, but restricted as much as possible to limit the difficulty of the search [1]. Specifically to our project, the DSL needs to be linear so as to cooperate with the beam search algorithm.

Following this observation, we decided to construct our own DSL in order to restrict search space and simplify the synthesis process.

Our DSL is a stack-based concatenative language, this section will introduce you the concept of stacks, stack-based programming languages, concatenative, and the advantages of our DSL and the description of its commands.

2.3.1 Stack-based language

To understand stack-based language, we first need to understand the concept of a stack.

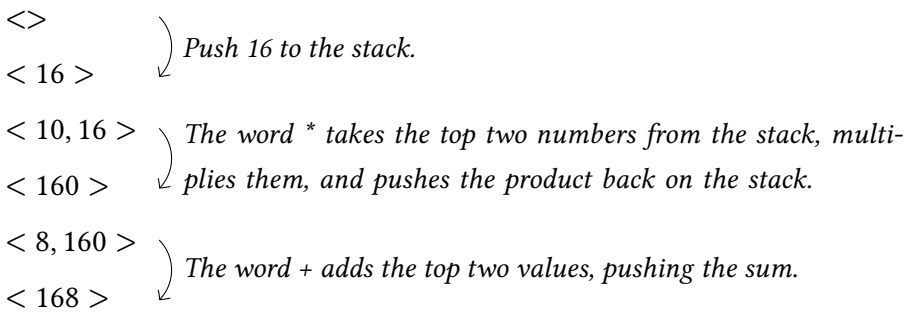
The stack is called that way because it resembles a stack in real life. Like a stack of plates, adding or removing is only possible at the top. They are two main principal operations of a stack as an abstract data type:

- Push: adds a new element to the top of the stack.
- Pop: removes the top element from the stack.

A stack based language works by having a global stack that the code implicitly works with. This means that you don't pass parameters when you call functions: all functions work by pushing and popping onto the global stack. Languages like Factor [25], Forth, Joy [28] fit this description.

Here comes an example for stack-based algorithms. Using reverse Polish notation [2], we get the result of the mathematical expression $(16 * 10 + 8)$ this way:

You would write $16\ 10\ *\ 8\ +$, and the computer will evaluate from left to right. A value will be pushed onto the stack, a word will be called, assuming the leftmost is the top of the stack. The stack will change as the following when the program is running.



2.3.2 Concatenative and stack languages

Even though the terms stack language and concatenative language sometimes get thrown around interchangeably, they actually represent similar but distinct classes of languages [3].

"Concat is a functional language (no explicit states) with static types and type inference. A concatenative language can also be dynamically typed and work without type inference; some variants are also functionally impure." [16] Concatenative languages are where the main way to build programs is by composing functions which all manipulate a single object. While stack-oriented languages operate on one or more stacks. They consider data of each stack, by utilising one piece of data from atop the stack, and returning data back atop the stack.

2.3.3 Stack-based concatenative language

Our DSL was largely inspired by Cat programming language [10] which is a statically typed stack-based pure functional language. A program in our DSL is a sequence of terms, where each term can change the state of the global stack.

The figure below shows the syntax of the stack-based concatenative language. The Instruction enclosed in square brackets represents a quotation. A value can be either an integer or a quotation and a term is either a value or an instruction. A program consists of terms which are used to operate the stack. See Section 2.3.4 for a detailed explanation for each instruction.

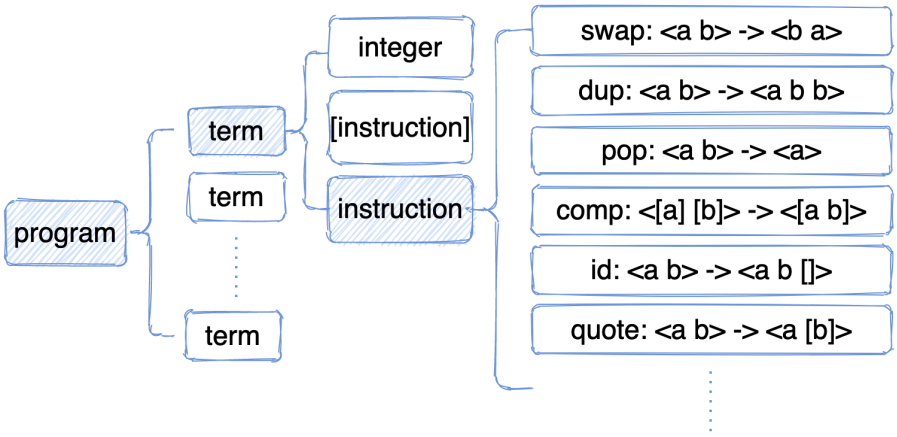


Figure 2.1: Syntax

There are many advantages of our language. Firstly, it is statically typed which means it is easy to filter out bad programs. Secondly, it does not have variables, there is only a list of operations where every operation can manipulate the global stack. Each operation is either a command or a value. All commands in this DSL are functions that take a stack as input, and return a new stack as output. This feature makes the language easy to be generated and encoded, this serves as a great advantage regarding the program synthesis task. Due to the linearity of this language, it is easier to use beam search on it than on other languages. Lastly, in support of higher-order functions, we have a special expression form - quotation. It is a unary constructor, an instruction can be turned into a value by quoting, then it can be pushed into the global stack and applied later. In our DSL the quotation of a program is written by enclosing it in square brackets.

Here is an example: given input stack $< 3, 5, 1 >$ and program $[swap, pop, mul]$, assuming the leftmost is the top of the stack.

$$
\begin{aligned}
&< 3, 5, 1 > \\
&\quad\downarrow\ Swap \\
&< 5, 3, 1 > \\
&\quad\downarrow\ Pop \\
&< 3, 1 > \\
&\quad\downarrow\ Dup \\
&< 3, 3, 1 > \\
&\quad\downarrow\ Mul \\
&< 9, 1 >
\end{aligned}
$$

In the end, we have 9, 1 in the resulting stack.

2.3.4 Descriptions of stack-based concatenative language

During the animated demonstration, you may encounter the following instructions, below are some detailed explanations.

Swap For a stack with at least two items, interchange the top two elements.

Dup For a stack with at least one item, duplicate the topmost element on the stack.

Pop For a stack with at least one item, remove the topmost element from the stack.

Comp For a stack with two quoted programs on the top, replace them with a new quotation that is the result of composing the top quotation with the second quotation.

Id For any stack, push a quoted identity program to the stack.

Quote For a stack with at least one item, turn an integer or an instruction to quotation form.

Apply For a stack with at least one quotation, apply takes the top quoted program, and applies it to the rest of the stack.

Dip For a stack with at least one quotation, apply the quoted program under the top most element to the rest of the stack.

Add For a stack with at least two integers, add the top two integers.

Sub For a stack with at least two integers, subtract the Second integer from the first integer.

Mul For a stack with at least two integers, multiply the top two integers.

Div For a stack with at least two integers, divide the Second integer with the first integer.

Mod For a stack with at least two integers, get the remainder of the second integer divided by the first integer.

Overall, our DSL contains:

- First-order instructions Swap, Dup, Pop, Comp, Id, Quote, Apply, Dip.
- Integer instructions Add, Sub, Mul, Div, Mod.

By converting them into quotation form, we can use them as higher-order commands.

2.4 Using

2.4.1 System functions

After successfully running the program, you will see the following main screen. It has a brief description of the software and some instructions to the user.

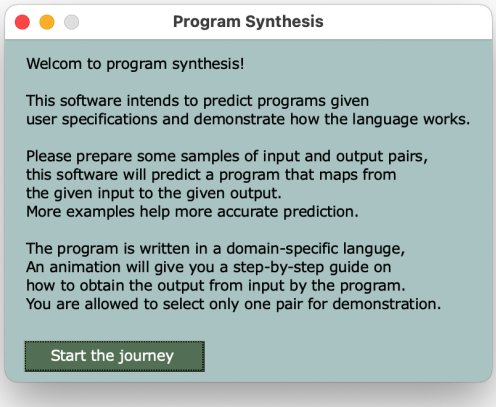


Figure 2.2: Main screen

The button leads to the window of user specification (Figure 2.3), you can type the number of input and output pairs that you want to give. These pairs will later be sent to the beam search algorithm to make the prediction and one of them will be selected for demonstration.

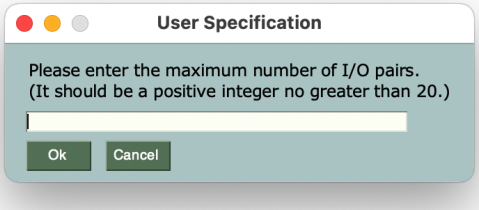


Figure 2.3: User Specification

Then you will be redirected to another window for entering concrete samples. You need to pay attention to the requirements written in the windows box, if you give ille-

gal values, you will get error messages(We will discuss this later in Section 2.4.2). For example it is six in this case (see Figure 2.3).

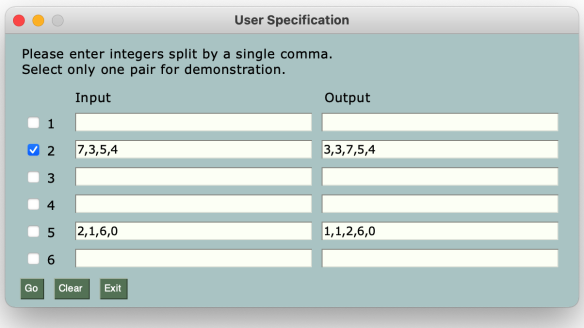


Figure 2.4: User Input

By clicking the "Ok" button, you will be redirected to another window. Here the user needs to enter at least one pair and at most six pairs of input and output examples. More pairs provide the model with more information, so the result will be more accurate. After finishing the input, you need to choose only one pair among what you have typed for demonstration purpose (as shown in Figure 2.4). After specifying all the needed information, the model will work on generating the program to satisfy user input, you will see a screen similar to Figure 2.5. The maximum length of the progress bar is set to equal to the max depth of beam search. It is possible that the suitable program is found before the searching reach the deepest level. Therefore, the screen may end before the progress bar reaches the end.

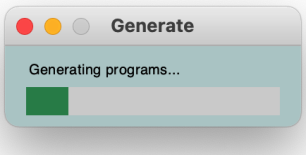


Figure 2.5: Progress Bar

When beam search gets the desired program, you will the the screen as Figure 2.6 shows. On the top of the main screen displays the instructions that can finish the given task. The stack below it starts with the initial state and will change according to the instruction. User can click the "Next" button to see how each step works and how the stack changes to go from the input to output with the generated instructions. And the checkbox can be used to hide or show the user input.

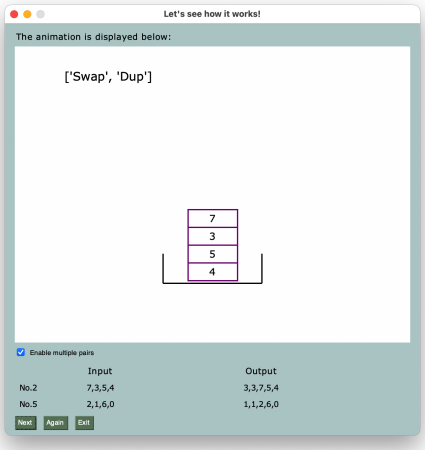


Figure 2.6: Demonstration screen

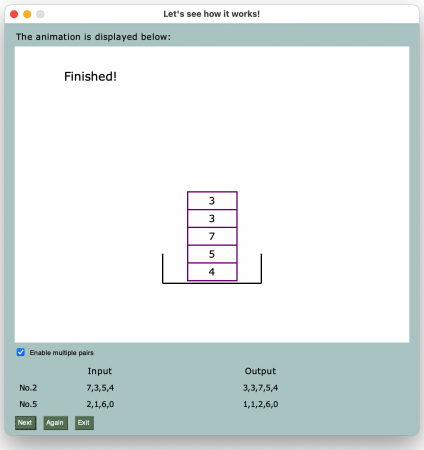


Figure 2.7: Finish screen

If the search ends, but no program is found, then the user can try again with other input and output pairs.

2.4.2 Run-time system messages

At the beginning, when the user is required to give a maximum number for input and output pairs, if they press the "Ok" button without entering an integer or entering other illegal characters which do not meet the requirement of the description, they may get the following error messages.

Figure 2.8: Error message for empty input

Figure 2.9: Error message for negative numbers

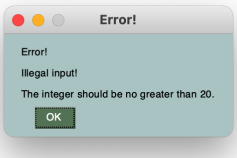


Figure 2.10: Error message for large numbers

After specifying the custom I/O pairs, the user needs to tick one checkbox for demon-

stration. They may get the following error messages if:

- They forget to select one pair to demonstrate. See Figure 2.11.
- They select more than one pair. See Figure 2.12.

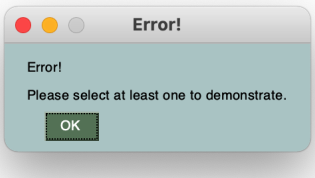


Figure 2.11: Error message for not selecting

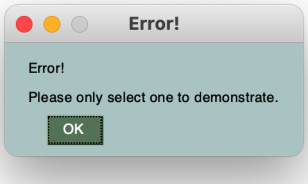


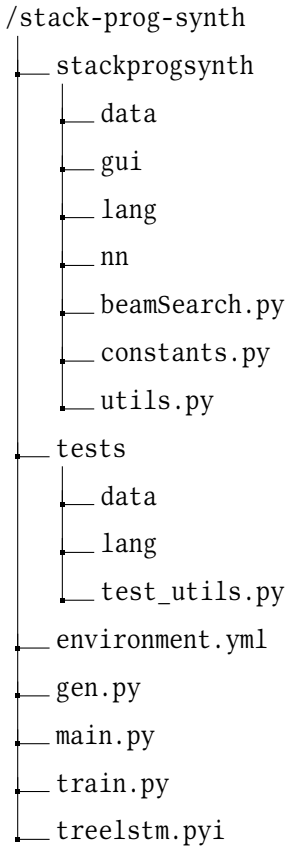
Figure 2.12: Error message for multiple selecting

Chapter 3

Developer Documentation

3.1 Overview

3.1.1 File structure



The tree above shows the file structure of the whole project. This is a joint work of a fellow student Xie Zongpu and me.

`stack-prog-synth` is the project folder and `stackprogsynth` is the source folder, the following was written by me:

1. `lang` folder, except `type.py`: for data generation, encoding, and program evaluation.
2. `nn` folder, except `data.py`: for building the model and measuring the accuracy.
3. `gui` folder and `main.py` file: for graphical user interface.
4. `constants.py` file: storing constant parameters for synthetic data generation and network construction.

The figure below shows the dependencies of files on my part. `main.py` is in the root folder `stack-prog-synth` while all other paths are relative to the source folder which is `stackprogsynth`. The file `beamSearch.py` was written by my colleague, however, it is necessary to be included in the figure since this part serves as an important bond between different modules.

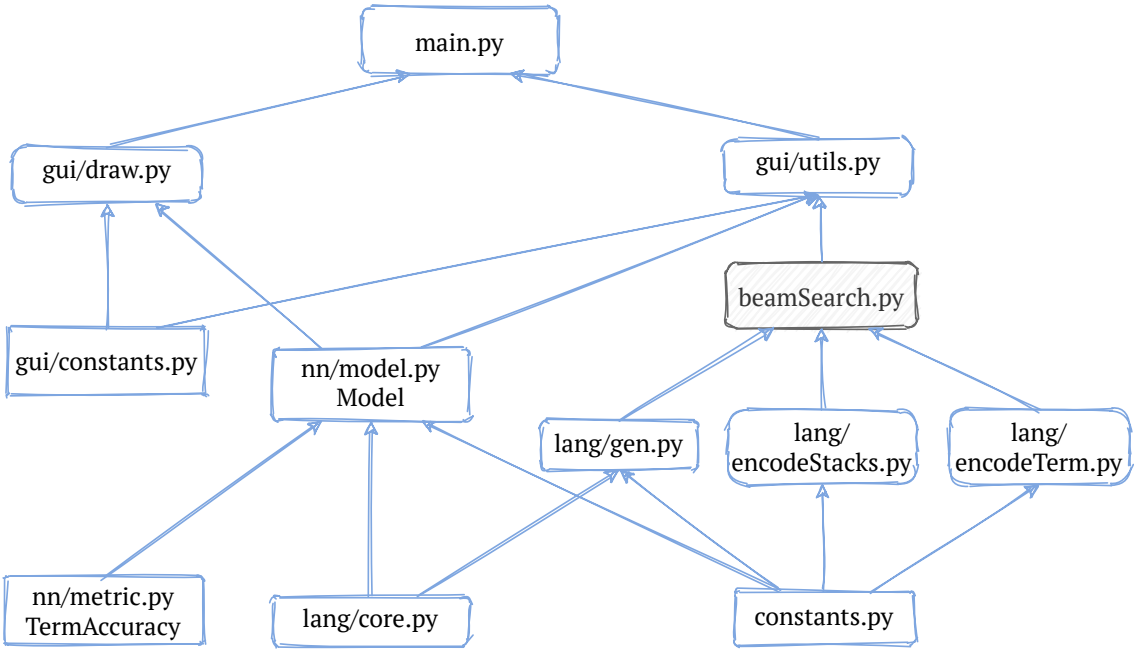


Figure 3.1: Dependencies of modules

The next session will give a brief summary of each module and the following sections will explain the implementation of each part in detail.

3.1.2 Software architecture

Figure 3.2 is the class diagram that covers most part of the project. It shows the connections between separate modules.

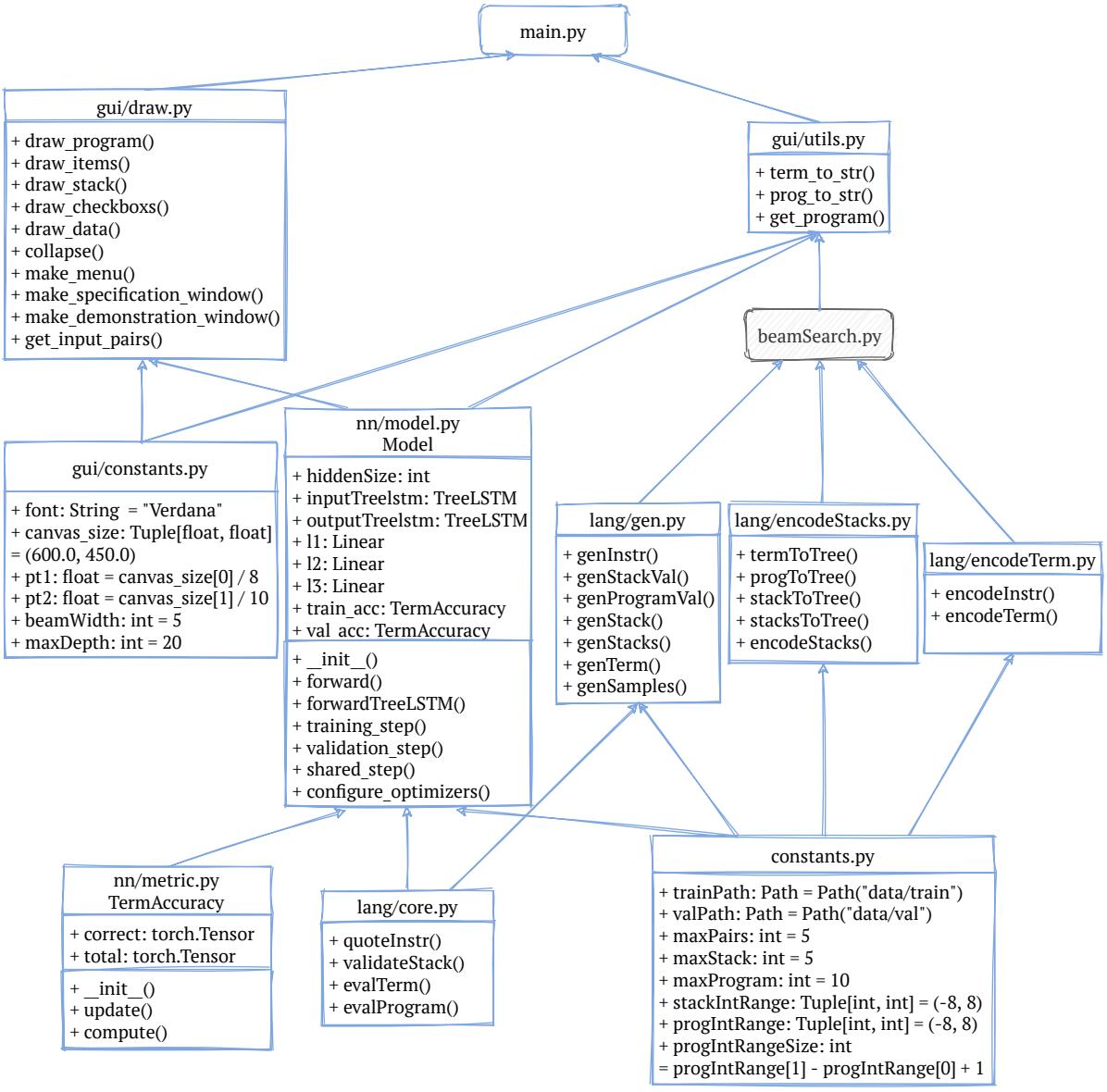


Figure 3.2: Connections between different modules
(function parameters and return types are omitted)

To start with, the user can run `main.py` to start the program. It has the main function that uses PySimpleGUI [22] to render windows and handle events. Coordinating with

draw.py and utils.py, it is able to pass user input to the beam search algorithm and get the predicted result.

File draw.py offers functions for stack animations and layout design. In utils.py, get_program will give trained model to beamSearch to obtain the prediction, and prog_to_str will convert the program to string so that it can be displayed in the window. constants.py in gui folder has constant parameters for rendering the window and using the beam search algorithm.

model.py has the class Model. It inherits LightningModule from the PyTorch Lightning [13] library. This custom class has methods to build the network, create Tree-LSTM and linear layer, and train the model. It also has instances of TermAccuracy to measure the performance of the prediction.

gen.py is used for synthetic data generation, this will be discussed in detail in Section 3.3. Files encodeStack.py and encodeTerm.py are used to convert data structures to tree then encode them, so that the Tree-LSTM layer can be applied. Section 3.3.3 will talk more about this.

The file core.py, as the name suggests, contains the core of the language. It implements the stack-based concatenative languge, see details in Section 3.2. The evaluation functions evalTerm and evalProgram are used to get the target result from the input. validateStack checks the integer ranges in the stack and quoteInstr converts an instruction to a quotation.

In the root folder, constants.py defines constant arguments for data synthesis and layer creation. It contains integer ranges that are used by the encoding files for validation. And the stack size and program length are used by gen.py for generating stack values and programs.

3.1.3 General workflow

The following figure gives an overview of the whole project. The part inside the rectangle is the architecture of our neural network, and this will be discussed in Section 3.4.

Generally speaking, we generate random input stacks and programs, by running the evaluation function evalProgram in core.py, we can get the corresponding output stacks. We then encode the input and output stacks and feed them into the Tree-LSTM neural network. Using supervised learning, we encoded the desired output program at the same time and compare it with the result that is given by the model. After this, the neural network will calculate the loss and the Adam optimizer [19] will make mini-batch gradient descent and back-propagation [26], repeat the training process until the model

reaches a decent validation accuracy.

When the model is prepared, we can use beam search on it to predict terms. Given input and output pairs, our trained model will give a probability for each term. The beam search algorithm then will rank the scores and find the most suitable fit as a solution.

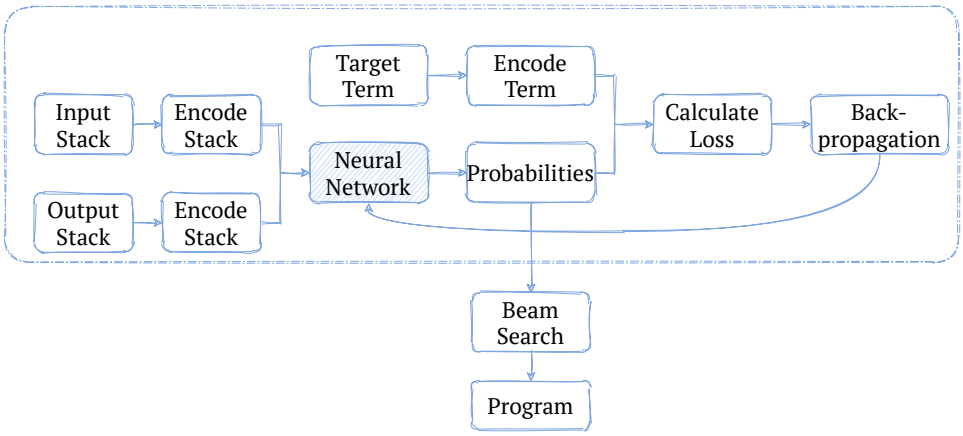


Figure 3.3: Workflow

3.2 DSL in Implementation

The following shows the UML for implementation of the stack-based concatenative language.

`Instr`, `Val`, `Int` and `Quoted` are classes for the ADT constructor. `Term`, `Program`, `Instruction` and `Value` are type synonyms, with `Term` defined as `T.Union[Instr, Val]`, `Instruction` defined as `T.Union[ Swap, Dup, Pop, Comp, Id, Quote, Apply, Dip, Add, Sub, Mul, Div, Mod ]` and `Value` defined as `T.Union[Quoted, Int]`.

Type `Union` in the typing module [14] was used to in the procedure. `Union[X, Y]` mean either X or Y. Similar to inheritance [5] in object-oriented programming [6], it represents subtypes from base types. However, it is closed for extensions [7].

`Program` is the type synonym of `Deque[Term]`. The abstract data type `Deque` gives back a new object instead of changing the original. This makes programs more efficient to manipulate persistently, meaning it allows access to any version, old or new, at any time [12]. When using the beam search algorithm, we need to be able to append to one version of the structure in multiple ways so as to check different possible terms.

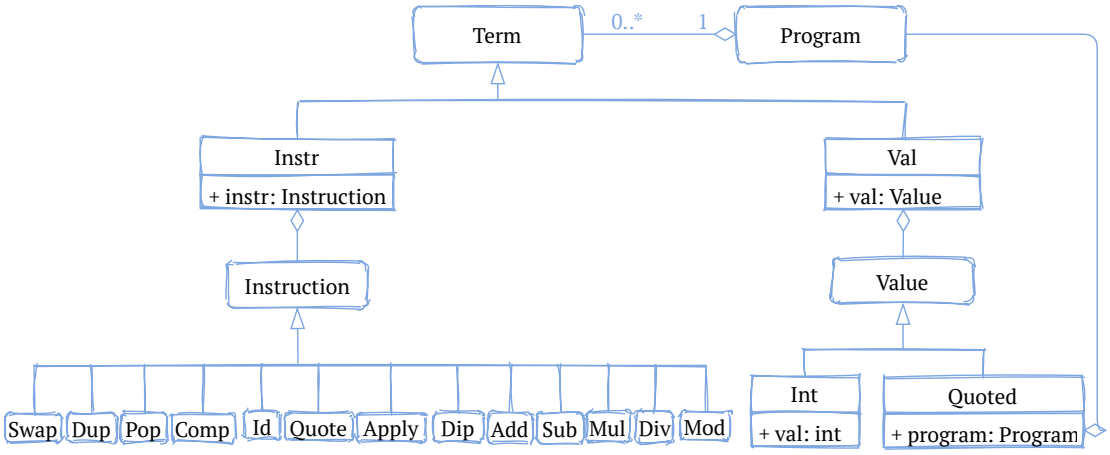


Figure 3.4: Class diagram of the stack language AST

Section 2.3.4 has a detailed description of each instruction and `evalTerm` implements the functionalities of each of them. Values are pushed to the stack. Integer operations `Add`, `Sub`, `Mul`, `Div` and `Mod` expect two integer values on the top of the stack. Some functions expect quoted programs on top of the stack and execute them in many different ways, effectively by dequoting, examples are `Apply` and `Dip`. The code below shows a small part of evaluating a term on a stack in `evalTerm` function.

```
        if isinstance(term.instr, Comp):
            fst, stack = stack.pop()
            snd, stack = stack.pop()
            if isinstance(fst, Quoted) and isinstance(snd, Quoted):
                return stack.push(Quoted(deque(snd.program + fst.program)))
            raise ValueError(fst, snd)
        if isinstance(term.instr, Id):
            return stack.push(Quoted(deque()))
        if isinstance(term.instr, Quote):
            value, stack = stack.pop()
            return stack.push(Quoted(deque([Val(value)])))
```

Listing 3.1: Implementation of Comp, Id, and Quote

3.3 Data Processing

3.3.1 Synthetic data generation

Synthetic data is artificial data generated to preserve privacy, testing systems, or creating training data for machine learning algorithms. Synthetic data generation is critical since it is an important factor in the quality of synthetic data [11]. If one does not have clean and well-prepared data, they may encounter garbage in, garbage out situations. Besides, the performance of a machine learning model is upper bounded by the quality of the data [17]. Therefore, generating enough amount of data with good quality is not only a prerequisite but also one of the requirements for a successful model training.

For our project, we need to generate stacks and programs separately. We treated a tuple of an input stack and a term and the resulting stack as one sample. Figure 3.5 illustrates the functions we wrote for synthesizing data and the relationship between them. And here follows how we generated samples in detail.

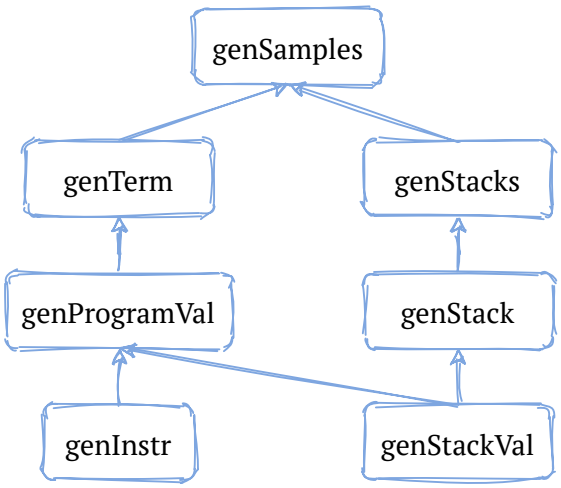


Figure 3.5: Functions for data generation

For programs, we have `genProgramVal` which creates random integer values in a predetermined range and uses `genInstr` to generate instructions. The `genTerm` function gets a random stack, returns a tuple of a generated term, and a tuple of new stacks. It will prune away invalid stacks or redundant programs.

For stacks, `genStack` uses `genStackVal` to get random integer values within a certain bound, for the sake of simplicity, we now only allow input stacks to have integers. `genStacks` returns a tuple that contains `maxPairs` of stacks, and each stack has a length that equals to `maxStack`. Finally, `genSamples` (see Listing 3.2) uses `genTerm` and `genStacks` to return a list of samples. `genTerm` checks the type of the new term and the elements in the new stacks. `genSamples` managed to avoid repetition of stacks in the resulting list.

```
def genSamples(
    rng: Random = Random(),
) -> T.List[T.Tuple[T.Tuple[Stack[Value], ...], T.Tuple[Stack[Value], ...], Term]]:
    samples: T.List[T.Tuple[T.Tuple[Stack[Value], ...], Term]] = []
    currStacks = genStacks(rng)
    inStacks = currStacks
    progNum = rng.randint(1, maxProgram)
    result = {currStacks}

    for _ in range(progNum):
        newTerm, newStacks = genTerm(inStacks, rng)
        if newStacks not in result:
            result.add(newStacks)
            samples.append((currStacks, newTerm))
            currStacks = newStacks
    return [(inp, currStacks, newTerm) for (inp, newTerm) in samples]
```

Listing 3.2: genSamples function

It is important to note in advance that integers are infinite, we have to set bound to them in order to do the encoding before passing them to the neural network. All the predetermined ranges are written in `constants.py` file. We set the range of integers in the program from -8 to 8, since small numbers are more frequent and a program with small numbers can manipulate big numbers. The file also defines the maximum size of a stack and the maximum allowed number of input-output pairs in a stack. These settings will not restrict our program to a large extent since integers can always get bigger numbers by adding or multiplying and stacks can expand.

3.3.2 Abstract syntax tree embedding

By having quotations in our stacks, they can have programs as first-class functions, which makes them not linear structures anymore. As a result, we need to look for neural networks that are suitable for tree-structured inputs instead of using the conventional recurrent neural network which is for linear data. We decided to use Tree-structured Long Short-Term Memory [27] networks to deal with non-linearity. Tree-LSTM is a generalization of LSTM which works on tree-structured inputs. We used a Python library `pytorch-tree-lstm` [8]. It is an implementation of the child sum Tree-LSTM model [27],

and it supports vectorized tree evaluation and batching.

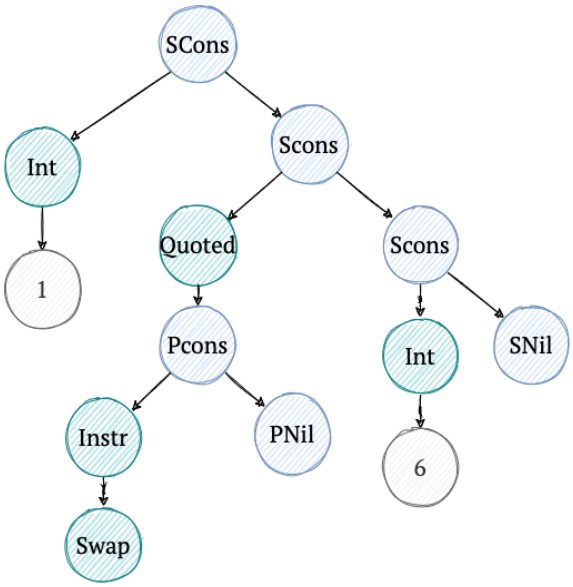


Figure 3.6: Abstract syntax tree

First, we need to convert stacks to trees. As the example shown in Figure 3.6, by having constructors like SCons, SNil, PNil, Quoted (they are stored in `labelClasses`, see Listing 3.3), we can turn the stack $< Int(1), Quoted([Instr(Swap)]), Int(6) >$ to a tree. Then, with the help of the library, we are able to convert each node to a tensor so that it can be given to the Tree-LSTM structure.

```
labelClasses: T.List[T.Union[str, int]] = (
    [
        constr.__name__
        for constr in T.get_args(Term) + T.get_args(Value) + T.get_args(Instruction)
    ]
    + ["PCons", "PNil", "SCons", "SNil", "TCons", "TNil"]
    + [i for i in range(stackIntRange[0], stackIntRange[1] + 1)]
)
```

Listing 3.3: labelClasses

3.3.3 Encoding

From Figure 3.3 we can deduce that it is necessary to encode stacks and target terms before training.

Figure 3.7 exhibits the functions written for encoding stacks, the names are self-explanatory. In order to apply the Tree-LSTM layer on the stacks, it is important to first convert them to trees before encoding. As we saw in Figure 2.1, programs comprise terms and terms consist of values, and quoted programs can be values. Hence, the functions `termToTree`, `progToTree` and `valToTree` are mutually recursive, it is like normal recursion where a function uses itself, they instead use each other. Mutual recursion is very common in functional programming.

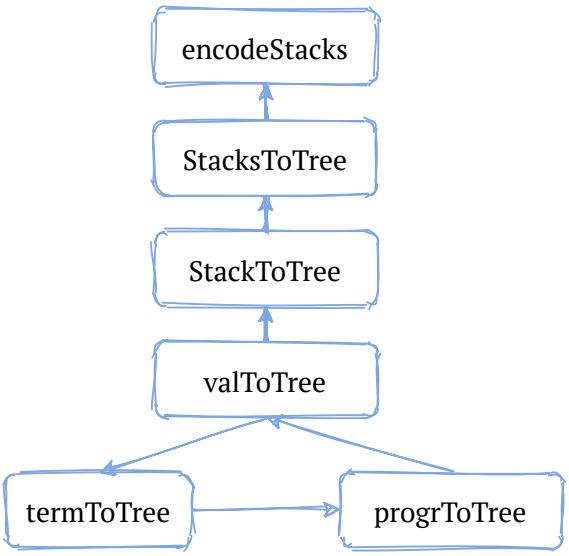


Figure 3.7: Functions for encoding stacks

The listings 3.4 and 3.6 are code for encoding instructions and terms. For instructions encoding, we return one plus the corresponding index of the given instruction. Since the 0th place is reserved for non-instruction types.

```
def encodeInstr(instr: Instruction) -> int:
    return T.get_args(Instruction).index(instr.__class__) + 1
```

Listing 3.4: Function for encoding an instruction

```
class EncodedTerm:
    termLabel: int
    instrLabel: int
    integer: int
```

Listing 3.5: EncodedTerm class

`encodeTerm` returns a tuple of type `EncodedTerm`, it is a class that we created to store the encoding result, as the Listing 3.5 shows. It has three fields:

- `termLabel`: Indicating the type of the new term, 0 for instructions, 1 for quotations, and 2 for integers.
- `instrLabel`: Specifying the type of the instruction, 0 for integers.
- `integer`: Representing the integer value, 0 for non-integer types and 1 to predetermined `progIntRangeSize` for integers.

In `encodeTerm`, the `isinstance` function returns True if the specified object is of the specified type, otherwise False. We have two validations here:

- Whether the quotation has only a single instruction.
- Whether the integer value within the predefined range.

A `ValueError` will be raised if the validation fails, otherwise, it returns an `EncodedTerm` instance.

```
def encodeTerm(term: Term) -> EncodedTerm:
    if isinstance(term, Instr):
        return EncodedTerm(0, encodeInstr(term.instr), 0)
    if isinstance(term, Val):
        if isinstance(term.val, Quoted):
            prog = term.val.program.left()
            if isinstance(prog, Instr):
                return EncodedTerm(1, encodeInstr(prog.instr), 0)
            raise ValueError(prog)
        if isinstance(term.val, Int):
            intVal = term.val.val
            if intVal < progIntRange[0] or intVal > progIntRange[1]:
                raise ValueError("Integer value out of range.")
            return EncodedTerm(2, 0, intVal - progIntRange[0] + 1)
```

Listing 3.6: Function for encoding a term

3.4 Network Construction

In this section, we will first give a holistic view of the network structure, then explain the implementation of network building, layer creating, and model training in detail. And hyper-parameters are listed in the end.

3.4.1 Network architecture and classes

Figure 3.8 shows the architecture of our neural network. We mainly used the `PyTroch` [24] library to build the neural net. The input of it is input and output stacks of a single program. Following the steps described in Section 3.3.2, we encode the stacks and feed them into Tree-LSTM networks.

In the hidden linear layers, the model learns and improves by evaluating the input and comparing it with the target output. We wrote a custom class to score the output and this will be discussed in Section 3.4.5.

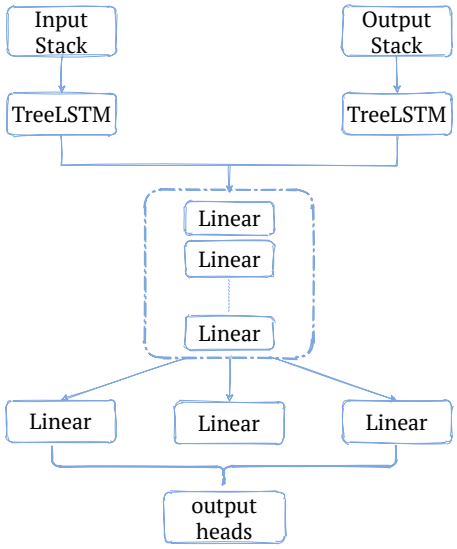


Figure 3.8: Network architecture

In this stage, the `pytorch-tree-lstm` [8] library helps the model increase efficiency in evaluating. Usually, efficient batching of tree data is complicated. It needs to evaluate all of a node's children before it can evaluate the node itself. To minimize the performance impact of this issue, the library breaks the node evaluation process into steps. In this way, each step it evaluates all nodes for which all child nodes have been previously evaluated. This allows the user to evaluate multiple nodes with each torch operation, increasing computation speeds by an order of magnitude over recursive approaches.

Take Figure 3.6 for example, on the first step of the tree calculation, the library can evaluate nodes 1, Swap, 6, PNil and SNil in parallel as they are all leaves of the tree, i.e. none of them has any child nodes. At the second step, it will evaluate node Int, Instr, as their child nodes were evaluated previously. Finally, it evaluates node 0, which depends on nodes 1 and 2. By doing this, the library can reduce a 13-node computation to six steps. As the dataset gets larger and trees get bigger with more leaf nodes, the user can experience larger performance gains.

The network's output heads finally produce 3 vectors, corresponding to the three fields of `EncodeTerms` explained in Section 3.3.3

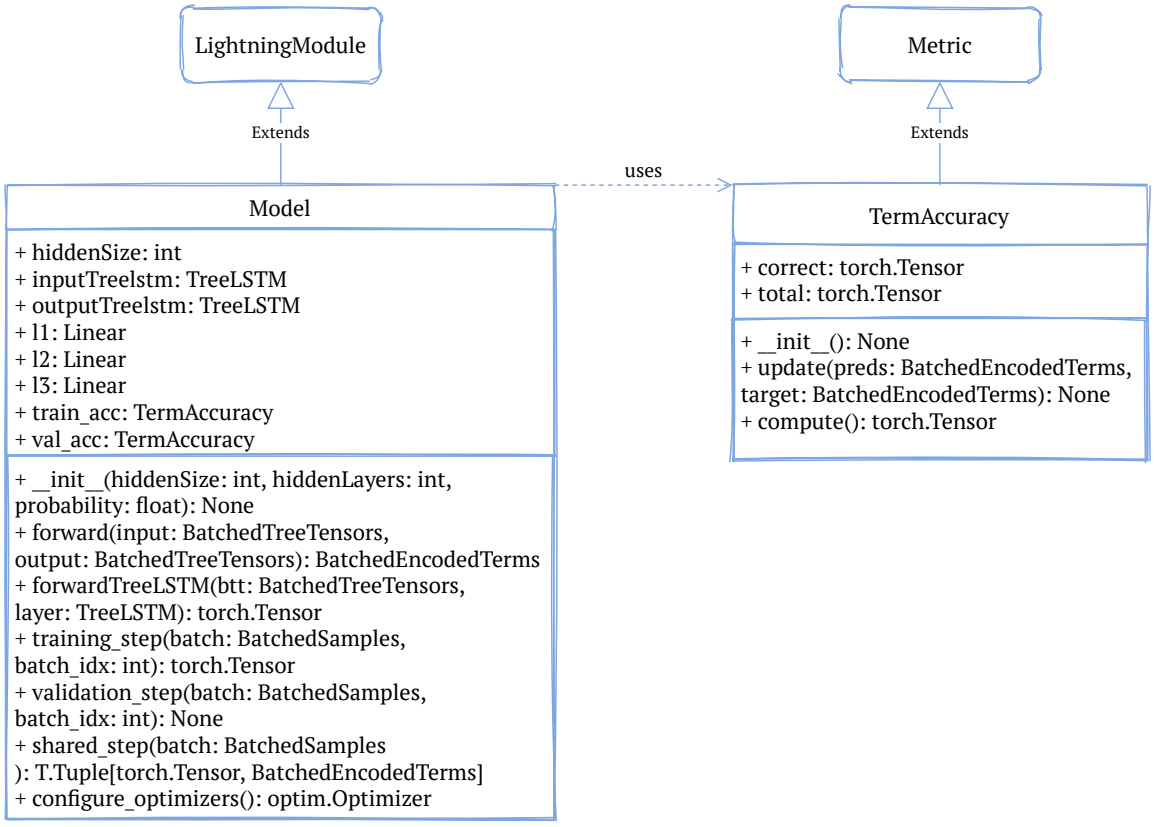


Figure 3.9: Model class and TermAccuracy class

The image above is a class diagram for `Model` and `TermAccuracy`. They extends classes `LightningModule` and `Metric`, both of them are from the PyTorch Lightning [13] library. Section 3.4.2 to Section 3.4.4 goes through all the functions in the `Model` class, and Section 3.4.5 talks the class `TermAccuracy` in detail.

3.4.2 Building the network

Having seen the structure of the whole network, we are now going to see the implementation of it.

Listing 3.7 shows the constructor of the model. We have the number of nodes in each middle layer (`hiddenSize`), the number of hidden layers (`hiddenLayers`).

```
def __init__(self, hiddenSize: int, hiddenLayers: int):
    super().__init__()
    self.save_hyperparameters()
    self.hiddenSize = hiddenSize
    # Input layers
    self.inputTreelstm = TreeLSTM(len(labelClasses), hiddenSize)
    self.outputTreelstm = TreeLSTM(len(labelClasses), hiddenSize)
    # Hidden layers
    self.hidden = nn.ModuleList()
    for i in range(hiddenLayers):
        middle = nn.Linear(2 * hiddenSize, 2 * hiddenSize)
        self.hidden.append(middle)
    # Output layers
    self.l1 = nn.Linear(2 * hiddenSize, 1 + len(T.get_args(Value)))
    self.l2 = nn.Linear(2 * hiddenSize, 1 + len(T.get_args(Instruction)))
    self.l3 = nn.Linear(2 * hiddenSize, 1 + progIntRangeSize)

    self.train_acc = TermAccuracy()
    self.val_acc = TermAccuracy()
```

Listing 3.7: Model constructor

We first create two Tree-LSTM layers for input and output stacks, the size of each input sample is the number of all the constructors for the abstract syntax tree and the number of features of the output of both layers is the hidden size. We concatenated two Tree-LSTM layers together before going through linear layers. Hence, the input and output size of middle layers are twofold of `hiddenSize`. Three output layers correspond to the output heads elaborated in 3.4.

3.4.3 Applying layers

Here follows how the input is processed through the network. We have `forwardTreeLSTM` and `forward` to do this job.

`forwardTreeLSTM` is used to apply Tree-LSTM layers to inputs. Four tensors `features`, `adjacency_list`, `node_order`, and `edge_order` store the encoded tree structure and features. Since the calculation of Tree-LSTM goes bottom up, we select the root nodes as the final result. The `unbatch_tree_tensor` is from the `pytorch-tree-lstm` library [8], it is used to unbatch the batched samples into a list. And we used `torch.unsqueeze` to create a new outer dimension to concatenate the samples.

```
def forwardTreeLSTM(self, btt: BatchedTreeTensors, layer: TreeLSTM) -> torch.Tensor:
    # Applying the layer.
    result, _ = layer(
        btt["features"].float(),
        btt["node_order"],
        btt["adjacency_list"],
        btt["edge_order"],
    )
    # Acquiring the calculated features of the root node for each sample.
    result = [
        torch.unsqueeze(sample[0], 0)
        for sample in unbatch_tree_tensor(result, btt["tree_sizes"])
    ]
    return torch.cat(result)
```

Listing 3.8: Tree-LSTM layers

In `forward` ,the comment `type: ignore[override]` is used to silence the type checker on this line caused by incompatible overrides. Here we first concatenate two Tree-LSTM layers together, then use a for loop to add enough number of middle layers to the module list in the constructor and use `relu` activation function on each of them. And finally, return a tuple of three output heads. The class `BatchedEncodedTerms` has the same fields as `EncodedTerm` in Listing 3.5, but the type of them is `torch.Tensor` instead of `int`.

```
def forward(  # type: ignore[override]
    self, input: BatchedTreeTensors, output: BatchedTreeTensors
) -> BatchedEncodedTerms:
    input1 = self.forwardTreeLSTM(input, self.inputTreelstm)
    input2 = self.forwardTreeLSTM(output, self.outputTreelstm)
    middle = torch.cat((input1, input2), 1)
    for i in range(len(self.hidden)):
        middle = self.hidden[i](middle)
        middle = nn.functional.layer_norm(middle, [self.hiddenSize * 2])
        middle = F.relu(middle)
    o1 = self.l1(middle)
    o2 = self.l2(middle)
    o3 = self.l3(middle)
    return BatchedEncodedTerms(o1, o2, o3)
```

Listing 3.9: Network layers

3.4.4 Training and improving model

`shared_step` uses cross-entropy to measure the loss of three output tensors from the forward layer. It returns a tuple of the sum of the loss and the predicted result, and this step is necessary for both training and validation stages. `training_step` and `validation_step` are basically the same except that they used different datasets.

```
def shared_step(
    self, batch: BatchedSamples
) -> T.Tuple[torch.Tensor, BatchedEncodedTerms]:
    predicted = self.forward(batch.input, batch.output)
    loss1 = nn.functional.cross_entropy(predicted.termLabel, batch.target.termLabel)
    loss2 = nn.functional.cross_entropy(
        predicted.instrLabel, batch.target.instrLabel
    )
    loss3 = nn.functional.cross_entropy(predicted.integer, batch.target.integer)
    return (loss1 + loss2 + loss3, predicted)
```

Listing 3.10: The common step

```
def training_step(
    self, batch: BatchedSamples, batch_idx: int
) -> torch.Tensor:  # type: ignore[override]
    loss, predicted = self.shared_step(batch)
    accuracy = self.train_acc(predicted, batch.target)
    self.log("train_loss", loss)
    self.log("train_acc", accuracy, prog_bar=True)
    return loss

def validation_step(
    self, batch: BatchedSamples, batch_idx: int
) -> None:  # type: ignore[override]
    loss, predicted = self.shared_step(batch)
    accuracy = self.val_acc(predicted, batch.target)
    self.log("val_loss", loss, prog_bar=True)
    self.log("val_acc", accuracy, prog_bar=True)
```

Listing 3.11: Training and validating

We used AdamW [19] as the optimizer. It has an adaptive learning rate, with default parameter starting from 0.001. This enables it to solve problem of slower convergence in the late stage of training.

```
    def configure_optimizers(self) -> optim.Optimizer:
        return optim.AdamW(self.parameters())
```

Listing 3.12: AdamW optimizer

3.4.5 Measuring the accuracy

Evaluating the machine learning algorithm is an essential part of any project. Accuracy is one metric to measure how often the algorithm classifies a data point correctly. It is the ratio of the number of correct predictions to the total number of input samples.

$$Accuracy = \frac{Numberof correct predictions}{Totalnumberof predictions}$$

The accuracy of our model is measured by the sum of the three tensors generated by

the output heads. We wrote our custom class for measuring it. `argmax` can find the class with the largest predicted probability.

```
class TermAccuracy(Metric):
    correct: torch.Tensor
    total: torch.Tensor
    def __init__(self) -> None:
        super().__init__()
        self.add_state("correct", default=torch.tensor(0), dist_reduce_fx="sum")
        self.add_state("total", default=torch.tensor(0), dist_reduce_fx="sum")

    def update(self, preds: BatchedEncodedTerms, target: BatchedEncodedTerms) -> None:
        t1 = torch.argmax(preds.termLabel, dim=1) == target.termLabel
        t2 = torch.argmax(preds.instrLabel, dim=1) == target.instrLabel
        t3 = torch.argmax(preds.integer, dim=1) == target.integer
        combined = torch.sum(t1 & t2 & t3)
        self.correct += combined
        self.total += torch.tensor(target.termLabel.numel())

    def compute(self) -> torch.Tensor:
        return self.correct.float() / self.total
```

Listing 3.13: Custom Accuracy class

3.4.6 Hyper-parameters and loss functions

Hyper-parameters are parameters of the structure of the neural network, they can control the process of learning. We train our models with the following parameters:

- Batch size: 32
- Middle layers: 5
- Hidden nodes: 1000
- Optimizer: AdamW
- Loss function: cross entropy
- Activation function: relu

3.5 Testing

An application without testing is not trustworthy. Automated tests can be very useful for killing bugs in application development. It also keeps validating the code while adding new code or refactoring old code.

There is a great number of functions that need to be tested in the project, and each function may need multiple test cases. Putting all the test cases into one `tests.py` file might not be a maintainable and scalable way. Therefore, we created a `tests` package in the root folder, and split tests into two folders `data` and `lang` to test files in the corresponding `data` and `lang` folder in the project. They also represent two main testings in this project. Pytest [18] is for unit testing in `data` folder and hypothesis test [20] is used for property testing in `lang` folder.

3.5.1 Unit tests

Python has built-in `unittest` module. It provides a solid base on which to build the test suite, but it has a few shortcomings. A number of third-party testing frameworks attempt to address some of the issues with `unittest`, and pytest [23] has proven to be one of the most popular. It is a mature and full-featured testing framework, from small tests to large-scale functional tests for applications and libraries alike.

```
/data
├── test_deque.py
├── test_llist.py
├── test_stack.py
└── test_tree.py
```

The structure above shows the files in `data` folder. With the powerful pytest library, we tested four main data structures: `deque`, `llist`, `stack`, and `tree`. For each data structure, we tested their main methods and the important functions related to them.

In `data/test_deque.py`, we have the tests cases for the methods of double-ended queue data structure. Here are two examples, they test the `appendleft` method and the `pop` method of deque.

```python
@given(st.lists(st.integers()), st.integers())
def test_deque_appendleft(xs: T.List[int], x: int) -> None:
    assert deque(xs).appendleft(x) == deque([x] + xs)

@given(st.lists(st.integers()).filter(lambda x: x))
def test_deque_pop(xs: T.List[int]) -> None:
    assert deque(xs).pop() == (deque(xs[:-1]), xs[-1])
```

Listing 3.14: Part of Deque testing

3.5.2 Property tests

Property-based testing has become quite famous in the functional world. It relies on properties which means that it checks that a function, program, or whatever system under test abides by a property. Most of the time, properties do not have to go into too many details about the output. They just have to check for useful characteristics that must be seen in the output.

Here is the content in the `lang` folder. For each function in `core.py`, `encodeStacks.py` and `encodeTerm.py` file, we gave one to three test cases.

```
/lang
├── test_core.py
├── test_encodeStacks.py
└── test_encodeTerm.py
```

The following are some examples in the `test_encodeStacks.py` file. For function `valueToTree`, we tested two types of the value - `Int` and `Quoted` separately. The same logic applies to `termToTree` as well, we tested two subtypes of the term, `Instr` and `Val`.

```python
def test_valueToTree_Int() -> None:
    assert valueToTree(Int(3)) == Tree("Int", [Tree(3, [])])

def test_valueToTree_Quoted() -> None:
    assert valueToTree(Quoted(deque([Instr(Add())]))) == Tree(
        "Quoted", [Tree("PCons", [Tree("Instr", [Tree("Add", [])]), Tree("PNil", [])])]
    )
```

Listing 3.15: Test valueToTree

```
def test_termToTree_Val() -> None:
    assert termToTree(Val(Int(8))) == Tree("Val", [Tree("Int", [Tree(8, [])])])

def test_termToTree_Instr() -> None:
    assert termToTree(Instr(Dup())) == Tree("Instr", [Tree("Dup", [])])
```

Listing 3.16: Test termToTree

For the function `encodeStacks`, the one-hot encoded result is tested. The `stack1` and `stack2` are predefined. For the return value, we check whether each row contains only one 1 and everything else is 0.

```
def test_encodeStacks() -> None:
    features = encodeStacks((stack1, stack2))["features"]
    for row in features:
        num = numel(row)
        cnt = bincount(row.int())
        assert cnt[0] == num - 1
        assert cnt[1] == 1
```

Listing 3.17: Test encodeStacks

For `progToTree`, `stackToTree` and `stacksToTree` in `encodeStacks.py` and `encodeInstr`, `encodeTerm` in `encodeTerm.py`, we calculate the desired result manually, and compare them with the function output. The following shows tests in `test_encodeTerm.py` file.

```
def test_encodeInstr() -> None:
    assert encodeInstr(Swap()) == 1

def test_encodeTerm() -> None:
    assert encodeTerm(Val(Int(3))) == EncodedTerm(2, 0, 12)
```

Listing 3.18: Test encodeTerm.py file

In `test_core.py` file, three test cases are given for each of `quoteInstr`, `validateStack`, and `evalProgram` function. And to test `evalTerm` function, we wrote test cases for each instruction.

3.5.3 Testing results

There are by far 60 tests altogether. We tested all the major functions, but we can always add more test cases for the same function for the sake of safety. The command to run all the tests in the `tests` folder is: `python -m pytest`. You will see the following results.

```
================================= test session starts ==================================
platform darwin -- Python 3.8.6, pytest-6.2.2, py-1.10.0, pluggy-0.13.1
rootdir: /Users/su/Desktop/stack-prog-synth
plugins: hypothesis-6.2.0
collected 60 items
tests/test_utils.py .                                                          [  1%]
tests/data/test_deque.py ..........                                            [ 20%]
tests/data/test_llist.py .                                                     [ 21%]
tests/data/test_stack.py .......                                               [ 33%]
tests/data/test_tree.py ..                                                     [ 36%]
tests/lang/test_core.py ......................                                 [ 73%]
tests/lang/test_encodeStacks.py ........                                       [ 86%]
tests/lang/test_encodeTerm.py ..                                               [ 90%]
tests/lang/test_type.py ......                                                 [100%]

================================= 60 passed in 5.96s ===================================
```

Chapter 4

Conclusion

The objective of this thesis was to design and implement software that can predict programs given user specifications and present to the user how the language works.

In the design phase, during the planning of the workflow, research was conducted on program synthesis, DSL, and Tree-LSTM networks. We chose Python as the programming language since Python code is understandable by humans, which makes it easier to build models for machine learning. And we decided to used PySimpleGUI to provide a graphical user interface. We found that it is more efficient and easier to work with if we construct our own domain-specific language.

In the implementation phase, we used many third-party tools to help us with the type checking and network building. We first implemented the DSL, then generated data with restrictions on integer ranges and validations on stack types, and encoded stacks. Then, we built the neural network, trained the model, and experimented with different hyper-parameters to improve the validation accuracy.

The user interface is clean and stylish. It is able to guide the user step by step of how the composition of terms do their magic when manipulating the stack. By presenting the animation of the stack, the user can be more familiar with the stack data structure, and get the gist of the stack-oriented language.

Overall, the project achieved the desired result with an informative user interface. It is a helpful software for users who want to know more about program synthesis and stack-based programming paradigm.

Future Work

The software is able to generate programs based on user specifications and provide a step-by-step demonstration to aid the user's understanding. But there are still many aspects that can be improved to be better.

From the implementation perspective:

1. In addition to `integers`, we can expand our language by including more types, such as `bool` and `list`.
2. Implementing more instructions to make our DSL a Turing language.
3. Improving our model with some hyperparameter optimization tools and introduce more complicated layers.
4. For greater efficiency of the search algorithm, we can introduce the state-of-the-art best-first beam search [21].

From the user perspective, we aim to develop more features to improve the user experience. For example, we plan to implement language translation. By converting our DSL to a lambda expression, then we can translate the lambda expression to other programming languages, such as python. This way, the user can export the language for later use.

List of Figures

List of Listings

Bibliography

[1] Matej Balog et al. "DeepCoder: Learning to Write Programs". In: *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017. url: https://openreview.net/forum?id=ByldLrqlx.

[2] Arthur W Burks, Don W Warren, and Jesse B Wright. "An analysis of a logical machine using parenthesis-free notation". In: *Mathematical tables and other aids to computation* 8.46 (1954), pp. 53–57.

[3] Dave Carlton. *Concatenative language*. url: https://www.concatenative.org/wiki/view/Concatenative%5C%20language.

[4] Wikipedia contributors. *Concatenative programming language*. 2021. url: https://en.wikipedia.org/wiki/Concatenative_programming_language.

[5] Wikipedia contributors. *Inheritance (object-oriented programming)*. 2021. url: https://en.wikipedia.org/wiki/Inheritance_(object-oriented_programming).

[6] Wikipedia contributors. *Object-oriented programming*. 2021. url: https://en.wikipedia.org/wiki/Object-oriented_programming.

[7] Wikipedia contributors. *Open–closed principle*. 2021. url: https://en.wikipedia.org/wiki/Open%5C%E2%5C%80%5C%93closed_principle.

[8] Jordan Dawe. *pytorch-tree-lstm*. 2019. url: https://github.com/unbounce/pytorch-tree-lstm.

[9] Arie Deursen, Paul Klint, and Joost Visser. "Domain-Specific Languages". In: *ACM SIGPLAN Notices* 35 (June 2000), pp. 26–36. doi: 10.1145/352029.352035.

[10] Christopher Diggins. *The Cat Programming Language*. 2007. url: http://www.cat-language.com/.

[11] Cem Dilmegani. *Synthetic Data Generation: Techniques, Best Practices & Tools*. 2021. url: https://research.aimultiple.com/synthetic-data-generation/.

[12] James R Driscoll et al. “Making data structures persistent”. In: *Journal of computer and system sciences* 38.1 (1989), pp. 86–124.

[13] WA Falcon and .al. “PyTorch Lightning”. In: *GitHub. Note: https://github.com/PyTorchLightning/pytorch-lightning* 3 (2019).

[14] Łukasz Langa Guido van Rossum Jukka Lehtosalo. *typing*. 2014. url: https://www.python.org/dev/peps/pep-0484/.

[15] Sumit Gulwani, Oleksandr Polozov, and Rishabh Singh. *Program Synthesis*. Jan. 2017. ISBN: 9781680832938. DOI: 10.1561/9781680832938.

[16] Dominikus Herzberg and Tim Reichert. “Concatenative programming-an overlooked paradigm in functional programming”. In: *International Conference on Software and Data Technologies*. Vol. 1. SCITEPRESS. 2009, pp. 257–262.

[17] Abhinav Jain et al. “Overview and Importance of Data Quality for Machine Learning Tasks”. In: Virtual Event, CA, USA: Association for Computing Machinery, 2020, pp. 3561–3562. ISBN: 9781450379984. DOI: 10.1145/3394486.3406477. url: https://doi.org/10.1145/3394486.3406477.

[18] Holger Krekel et al. *pytest 6.2*. 2004. url: https://github.com/pytest-dev/pytest.

[19] Ilya Loshchilov and Frank Hutter. “Decoupled weight decay regularization”. In: *arXiv preprint arXiv:1711.05101* (2017).

[20] David MacIver, Zac Hatfield-Dodds, and Many Contributors. “Hypothesis: A new approach to property-based testing”. In: *Journal of Open Source Software* 4.43 (Nov. 21, 2019), p. 1891. ISSN: 2475-9066. DOI: 10.21105/joss.01891. url: http://dx.doi.org/10.21105/joss.01891.

[21] Clara Meister, Ryan Cotterell, and Tim Vieira. “Best-First Beam Search”. In: *Trans. Assoc. Comput. Linguistics* 8 (2020), pp. 795–809. url: https://transacl.org/ojs/index.php/tacl/article/view/2169.

[22] Mike. *PySimpleGUI*. 2019. url: https://pysimplegui.readthedocs.io/en/latest/.

[23] Brian Okken. *Python testing with Pytest: simple, rapid, effective, and scalable*. Pragmatic Bookshelf, 2017.

[24] Adam Paszke et al. “PyTorch: An Imperative Style, High-Performance Deep Learning Library”. In: *Advances in Neural Information Processing Systems 32*. Ed. by H. Wallach et al. Curran Associates, Inc., 2019, pp. 8024–8035. url: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[25] Sviatoslav Pestov, Daniel Ehrenberg, and Joe Groff. “Factor: A dynamic stack-based programming language”. In: *Acm Sigplan Notices* 45.12 (2010), pp. 43–58.

[26] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. *Learning internal representations by error propagation*. Tech. rep. California Univ San Diego La Jolla Inst for Cognitive Science, 1985.

[27] Kai Sheng Tai, Richard Socher, and Christopher D. Manning. “Improved Semantic Representations From Tree-Structured Long Short-Term Memory Networks”. In: *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing of the Asian Federation of Natural Language Processing, ACL 2015, July 26-31, 2015, Beijing, China, Volume 1: Long Papers*. The Association for Computer Linguistics, 2015, pp. 1556–1566. doi: 10.3115/v1/p15-1150. url: https://www.aclweb.org/anthology/P15-1150/.

[28] Manfred Von Thun and R Thomas. “Joy: Forth’s functional cousin”. In: *Proceedings from the 17th EuroForth Conference*. 2001. url: http://www.complang.tuwien.ac.at/anton/euroforth/ef01/thomas01a.pdf.

[29] Amit Zohar and Lior Wolf. “Automatic Program Synthesis of Long Programs with a Learned Garbage Collector”. In: *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*. Ed. by Samy Bengio et al. 2018, pp. 2098–2107. url: https://proceedings.neurips.cc/paper/2018/hash/390e982518a50e280d8e2b535462ec1f-Abstract.html.