October 12, 2019

0.1 Grammar and Parser

I have a simple grammar just to accept limited number of pattern combinations and create a parse tree of it. I have used ANTLR (ANother Tool for Language Recognition) to generate the parser . The parser creates a tree of the input program making the outer-most skeleton as root. for example , let's say we pass a file having the following code :

Listing 1: example input program

```
a = Seq (10);
b = Seq (20);
main = Pipe(Farm(a),b);
```

The parser creates a tree with root Pipe pattern having two stages Farm and Sequential; Farm intrun will have one stage which is type of Sequential. After that i have used a breadth first search algorithm to construct a forest tree staring from the result of the parser. The tree construction is done by implementing rewriting rules at level of the skeletons;which means a tree can have multiple skeleton nodes and the visitor visits each node and creates a new tree from the result of the refactoring.

here are the basic rewriting rules:

- $\Delta \xLeftrightarrow[\text{Farm\_Intro}]{\text{Farm\_Elim}} farm(\Delta)$
- $Comp(\Delta_1, \Delta_2) \xLeftrightarrow[\text{Pipe\_Intro}]{\text{Pipe\_Elim}} Pipe(\Delta_1, \Delta_2)$
- $Map(\Delta) \xRightarrow[\text{Map\_Elim}]{} (\Delta)$
- $Comp(Comp(\Delta_1, \Delta_2), \Delta_3) \xLeftrightarrow[\text{Comp\_Assoc}]{} Comp(\Delta_1, Comp(\Delta_2, \Delta_3))$
- $Pipe(Pipe(\Delta_1, \Delta_2), \Delta_3) \xLeftrightarrow[\text{Pipe\_Assoc}]{} Pipe(\Delta_1, Pipe(\Delta_2, \Delta_3))$
- $Map(Pipe(\Delta_1, \Delta_2)) \xLeftrightarrow[\text{Pipe\_of\_map}]{\text{Map\_of\_Pipe}} Pipe(Map(\Delta_1), Map(\Delta_2))$
- $Map(Comp(\Delta_1, \Delta_2)) \xLeftrightarrow[\text{Comp\_of\_map}]{\text{Map\_of\_Pipe}} Comp(Map(\Delta_1), Map(\Delta_2))$

0.2 parser

ANTLR (Another Tool For Language Recognition) is parser generator that translates grammars to a parser/lexer in a target language, and the generated parser is used to construct skeleton tree.

Listing 2: grammar

0.3 refactoring algorithm

it starts by visiting the root node and proceeds to the child nodes . each visiting operation generates a set of trees ; they indicate different rewriting options for a particular tree.each newly created tree is inserted into queue so that it'll be refactored by the visitor. i have used breadth first search algorithm in combination with visitor pattern for the tree expansion. at the end we have a forest tree of patterns.

Listing 3: skeleton tree

```
Skeleton {
        Skeleton root;
        List<Skeleton> children;
        List<Skeleton> reWritngOptions;
        ReWringRule rule;
}
Edge{
        Skeleton from;
        Skeleton to;
        ReWritingRule rule;
}
```

Algorithm 1: refactoring algorithm

```
1  input: program to be parallelized
   Result: directed graph of the rewriting options
2  building skeleton tree st;
3  initialization;
4  add st to queue
5  while queue is not empty do
6      st =queue.remove();
7      patterns = refactor(st);
8      repeat
9          If pattern not in queue & pattern.height $\leq$ maxHeight
            queue.add(pattern);
10     until all patterns are inserted;
11     repeat
12         patterns = refactor (stage);
13         If pattern not in queue & pattern.height $\leq$ maxHeight
            queue.add(pattern);
14     until all stages are refactored;
15 end
```

line 1: input, the algorithm accepts input in format of Listing **??**.
line 2: building tree; the input code is parsed and a new parse tree is constructed .the parser generated by ANTLR parse and construct a tree from the input code.
line3: initialization; initlialize queue for the breadth first search, directed graph to hold the trees generated by the expansion process
line 4: start breadth first search
line 7-9: refactoring; this is implemented by using visitor pattern which visits every node of the tree staring from the root and each visit operation creates a new tree which in turn will be added into the queue for further refactoring. since this process creates infinite number of trees i have added a condition to stop the process. for a tree to be refactored it should have a height less than maxHeight. whenever a new alrernative rewriting option is constructed the algoithm will add it to the graph if it is not already present and it creates an edge from the original skeleton tree to the new one labeling it with the rewring rule used to generate it. if the tree already exists it will just create a new edge between the original tree and the the existing tree object with the rule.

line 10-14 is similar to above process except it works on the child nodes of the tree.

0.4 non functional parameters

i have focused mainly on the service time and parallelism degree of the skeletons. while creating trees of refactoring patterns is straight forward , allocating resources is a very difficult problem to tackle. here is how paralellism degree and service times are computed for each kind of skeleton:
let `Tscatter` T_s, `Tcollector` T_c ,`Temitter` T_e, `Tgather` T_g=1
$T_\triangle$ `is service time of the stage or worker skeleton`
`Sequential Skeleton: n = 1; Ts = ts`
`Farm Skeleton: the ideal parallelism degree is computed by` $\frac{T_e}{T_\triangle}$
`Pipeline Skeleton: the ideal parallelism degree is the number of stages`
`Comp Skeleton:`
`Map Skeleton: n =` $\sqrt{\frac{T_\triangle}{max(T_s,T_g)}}$
Farm:
the ideal service time is calculated using the formula: $\texttt{Max}(T_e, T_c, \frac{T_\triangle}{n})$
Pipeline:
$T_s = Max(T_{si}) \forall i \in pipestages$