November 24, 2019

## 0.1 Grammar and Parser

The grammar defines basic constructs of patterns and it is used as an input to parser generator ANTLR(ANother Tool for Language Recognition). ANTLR is used to generate a parser which translates input into a parse tree of skeletons. example input:

Listing 1: example input program

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

The parser creates a tree with root node Pipe skeleton having two stages Farm and Sequential; Farm stage will have one worker process which is type of Sequential. the resulting tree is passed to the refactoring process which is based on breadth-first search algorithm. the input tree is expanded into a forest of trees in the refactoring process. The tree expansion is done by implementing rewriting rules at level of the skeletons; a visitor based rewriter visits each skeleton node of a tree and creates a new tree by applying rewriting rules.

here are the basic rewriting rules:

- $\Delta \underset{\text{Farm\_Intro}}{\overset{\text{Farm\_Elim}}{\Longleftrightarrow}} farm(\Delta)$
- $Comp(\Delta_1, \Delta_2) \underset{\text{Pipe\_Intro}}{\overset{\text{Pipe\_Elim}}{\Longleftrightarrow}} Pipe(\Delta_1, \Delta_2)$
- $Map(\Delta) \underset{\text{Map\_Elim}}{\Longrightarrow} (\Delta)$
- $Comp(Comp(\Delta_1, \Delta_2), \Delta_3) \underset{\text{Comp\_Assoc}}{\Longleftrightarrow} Comp(\Delta_1, Comp(\Delta_2, \Delta_3))$
- $Pipe(Pipe(\Delta_1, \Delta_2), \Delta_3) \underset{\text{Pipe\_Assoc}}{\Longleftrightarrow} Pipe(\Delta_1, Pipe(\Delta_2, \Delta_3))$
- $Map(Pipe(\Delta_1, \Delta_2)) \underset{\text{Pipe\_of\_map}}{\overset{\text{Map\_of\_Pipe}}{\Longleftrightarrow}} Pipe(Map(\Delta_1), Map(\Delta_2))$
- $Map(Comp(\Delta_1, \Delta_2)) \underset{\text{Comp\_of\_map}}{\overset{\text{Map\_of\_Pipe}}{\Longleftrightarrow}} Comp(Map(\Delta_1), Map(\Delta_2))$

## 0.2 refactoring algorithm

The parser generates a tree structure with only a single node of type a tree which has skeleton patterns as its child nodes, this tree is fed into a refactoring algorithm which expands it into a tree of trees.The algorithm starting from the input tree performs tree expansion by implementing rewriting rules. the first tree is transformed into different trees and these trees are inserted as child nodes.the algorithm is based on breadth-first search and on the tree expansion process newly created trees are inserted into the queue . the algorithm is based on breadth-first search at the tree level and visitor pattern at the node level, which means trees are refactored in their insertion order and the refactoring process is done using the visitor pattern at nodes level. The visitor visits each nodes of a tree and generates new trees replacing the the visited node with it's alternate rewriting options. These new trees are inserted into queue of the algorithm and are processed in their insertion order. This process continues until the queue is empty and the new trees has height h (a tree can be inserted in queue only if it has height less than h). The tree is represented as adjacency list meaning a node can have n adjacent nodes which are it's alternative representations. The algorithm tries to explore all the possible options to represent a pattern tree in different combinations of patterns with out considering the performance metrics.

Listing 2: 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 ≤ maxHeight
   |  |   queue.add(pattern);
10 |  until all patterns are inserted;
11 |  repeat
12 |  |  patterns = refactor (stage);
13 |  |  If pattern not in queue & pattern.height ≤ maxHeight
   |  |   queue.add(pattern);
14 |  until all stages are refactored;
15 end
```

line 1: input, the algorithm accepts input in format of Listing 1.
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.

## 0.3 Non functional parameters

in the refactoring procedure the algorithm considers that there are enough number of resources to realize all the tree of patterns so the performance metrics are ideal cases. optimization instead uses the available resources and tries to allocate them in a way that will result a better performance and this resource scheduling process is a complex task. IBM ILOG CP Optimizer is used to tackle this problem. ILOG CP Optimizer is a constraint programming based tool that enables to model and solve scheduling problems. the model is comprised of an objective function and list of constraints . in this case the constraints are number of available resources and service time which means the number of resources allocated for each tree node and also the commutative resources assigned to them can not exceed the number of available resources. regarding the parallelism degree , it can not be more than the ideal one. thus the optimal parallelism is determined considering such constraints. the objective function is to minimize the service time , hence the solver gives the optimal parallelism degree that will result a minimum service time. Here is how parallelism 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
$T_\Delta$ is service time of the stage or worker skeleton
Sequential Skeleton: n = 1; Ts = ts
Farm Skeleton: the ideal parallelism degree is computed by $\frac{T_e}{T_\Delta}$ and the ideal service time is calculated using the formula: Max$(T_e, T_c, \frac{T_\Delta}{n})$
Pipeline: $T_s = Max(T_{si}) \forall i \in pipestages$
Map Skeleton: n = $\sqrt{\frac{T_\Delta}{max(T_s, T_g)}}$

However these formulas are used only to compute the ideal cases; whereas the optimal case is computed using the ILOG CP solver.