APPENDIX
ALGORITHMS FOR CFLREACHABILITY AND CHECKATTR

We briefly describe `CFLReachability` and `CheckAttr` in Fig. 12 and Fig. 13.

Fig. 12 shows the `CFLReachability` algorithm. The procedure `CFLReachability` operates similar to the shortest-path CFL reachability algorithm [10]. The procedure `CFLReachability` operates on the weighted modification graph, referred to as G. Additionally, `CFLReachability` utilizes a priority-based working queue, denoted as H, to store edges to be processed. The priority queue H is ordered by the weight of edges, where edges with lower weights are processed first. For edges with the same weight, root edges are processed last in the queue H to ensure the invariance of sub-problems in the dynamic programming. The queue H is implemented as a heap with a customized comparator.

When initially invoked, `CFLReachability` carries out a preparatory step, aiming to handle $A \rightarrow \epsilon$ production rules, as shown in Fig. 2a. Subsequently, the edges within G are enqueued into H to initialize the processing sequence.

The core of the algorithm is a looping construct that continues until the working queue H becomes empty. Within each iteration, the algorithm dequeues an edge e from H, capturing its associated label A. If A corresponds to the start symbol and the edge begins at the first vertex and ends at the last vertex, the algorithm promptly terminates, signifying the attainment of a valid path. Otherwise, `CFLReachability` systematically explores potential productions to reduce the current path. It does so by invoking three distinct subroutines: `try_production_1`, `try_production_2_left`, and `try_production_2_right`. These subroutines extend the exploration process, utilizing the edge's information and the graph G to attempt various production possibilities, as shown in Fig. 2b and Fig. 2c.

Different from original shortest-path CFL reachability algorithm, when inserting an edge with a higher weight into the graph, `CFLReachability` does not remove the edge with the same label and the same end vertex from the graph to avoid the loss of potential solutions.

Through this algorithm, `CFLReachability` endeavors to uncover viable paths that contain syntactically correct programs.

Fig. 13 shows the `CheckAttr` algorithm. The procedure `CheckAttr` performs attribute checking on the CFL reachability graph. The procedure `CheckAttr` checks an edge e_A with an inherited attribute value I_A and returns a set of synthesized attribute values $\mathbb{S}_A$.

To avoid repeated computation, `CheckAttr` utilizes a memoization map $\mathbb{M}$ to store the results of previous computations. If the result of the current computation is already in $\mathbb{M}$, `CheckAttr` returns the result directly.

Another optimization used in `CheckAttr` is to utilize a generator-like mechanism to return the synthesized attribute values. Instead of returning all the synthesized attribute values at once, `CheckAttr` returns an iterator that yields one synthesized attribute value at a time. This optimization allows `CheckAttr` to return the synthesized attribute values one by one, which is more efficient than returning all the synthesized attribute values at once.

For terminal edges, `CheckAttr` invokes `process_terminal` to compute the synthesized attribute value, where `process_terminal` is a user-defined function for computing the synthesized attribute value of a terminal edge. If `process_terminal` fails, `process_terminal` returns **null**, indicating the inherited attribute of the terminal does not conform to semantic constraints, so that `CheckAttr` skips the current value and continues to check the next attribute value.

For non-terminal edges, `CheckAttr` iterates over all the possible productions and invokes `process_left_inherited`, `process_right_inherited`, and `process_synthesized` to compute the inherited attribute value, the right inherited attribute value, and the synthesized attribute value, respectively. Here, `process_left_inherited`, `process_right_inherited`, and `process_synthesized` are user-defined functions for computing the inherited attribute value, the right inherited attribute value, and the synthesized attribute value of a non-terminal edge, respectively. Similarly, these functions return **null** if the attribute value does not conform to semantic constraints, so that `CheckAttr` skips the current value and continues to check the next attribute value.

The optimizations of `CheckAttr` are implemented as follows.

Pruning. `CheckAttr` only continues to check the attribute if the inherited attribute value conforms to semantic constraints. Otherwise, `CheckAttr` skips the current value and continues to check the next attribute value. This is implemented by returning **null** in `process_terminal`, `process_left_inherited`, `process_right_inherited`, and `process_synthesized` when the attribute value does not conform to semantic constraints.

Merging. `CheckAttr` stores a list of synthesized attributes in $\mathbb{S}_A$. The list $\mathbb{S}_A$ will remove duplicated synthesized attributes before returning. This is implemented by using an extra set to store the synthesized attributes in $\mathbb{S}_A$.

Memoization. `CheckAttr` stores the results of previous computations in $\mathbb{M}$. If the result of the current computation is already in $\mathbb{M}$, `CheckAttr` returns the result directly.

With the help of `CheckAttr`, OrdinalFix can perform attribute checking on the CFL reachability graph and find the shortest path that satisfies all the attribute constraints.

```
1: procedure CFLReachability(G, H)
2:     {G is the weighted modification graph}
3:     {H is the working queue with priority, which stores edges to be processed}
4:     if first time to call CFLReachability then
5:         for each rule A → ϵ, each vertex v_i do
6:             add ⟨v_i, v_i, A, 0⟩ to G
7:         end for
8:         for each edge e ∈ G do
9:             H.push(e)
10:        end for
11:    end if
12:    while H not empty do
13:        e = H.pop()
14:        A = e.label
15:        if A is the start symbol and e begins at v_0 and ends at v_n then
16:            return e
17:        end if
18:        try_production_1(e, G, H)
19:        try_production_2_left(e, G, H)
20:        try_production_2_right(e, G, H)
21:    end while
22: end procedure
```

Fig. 12. The `CFLReachability` algorithm.

```
1: procedure CheckAttr(e_A, I_A, 𝕄)
2:     {e_A is the edge to be checked}
3:     {I_A is the inherited attribute value}
4:     {𝕄 is the memoization map}
5:     {denote the symbol of e_A as A}
6:     if (e_A, I_A) is in 𝕄 then
7:         return 𝕄[(e_A, I_A)]
8:     end if
9:     construct empty list 𝕊_A to store results
10:    𝕄[(e, I)] = 𝕊_A
11:    if A is an edge with terminal symbol then
12:        S_A = process_terminal(A, I_A)
13:        if S_A is not null then
14:            𝕊_A.add(S_A)
15:            yeild S_A
16:        end if
17:    else
18:        for each edge expansion r for e do
19:            if r is in the form of e_A → ϵ then
20:                S_A = process_synthesized(A → ϵ, I_A, null, null)
21:                if S_A is not null then
22:                    𝕊_A.add(S_A)
23:                    yeild S_A
24:                end if
25:            end if
26:            if r is in the form of e_A → e_B then
27:                {denote the symbol of e_B as B}
28:                I_B = process_left_inherited(A → B, I_A)
29:                if I_B is not null then
30:                    𝕊_B =CheckAttr(e_B, I_B, 𝕄)
31:                    for each S_B ∈ 𝕊_B do
32:                        S_A = process_synthesized(A → B, I_A, S_B, null)
33:                        if S_A is not null then
34:                            𝕊_A.add(S_A)
35:                            yeild S_A
36:                        end if
37:                    end for
38:                end if
39:            end if
40:            if r is in the form of e_A → e_B e_C then
41:                {denote the symbol of e_B as B and the symbol of e_C as C}
42:                I_B = process_left_inherited(A → BC, I_A)
43:                if I_B is not null then
44:                    𝕊_B =CheckAttr(e_B, I_B, 𝕄)
45:                    for each S_B ∈ 𝕊_B do
46:                        I_C = process_right_inherited(A → BC, I_A, S_B)
47:                        if I_C is not null then
48:                            𝕊_C =CheckAttr(e_C, I_C, 𝕄)
49:                            for each S_C ∈ 𝕊_C do
50:                                S_A = process_synthesized(A → BC, I_A, S_B, S_C)
51:                                if S_A is not null then
52:                                    𝕊_A.add(S_A)
53:                                    yeild S_A
54:                                end if
55:                            end for
56:                        end if
57:                    end for
58:                end if
59:            end if
60:        end for
61:    end if
62: end procedure
```

Fig. 13. The CheckAttr algorithm.