CISC499

Compact and Interpretable Predictive Models for Linear Genetic Programming

Algorithm Final Draft

Linear Genetic Programming (LGP) basically is a single sequence of instructions. This algorithm is supposed to be used to detect non-effective code, delete and simplify the code without affecting the output, and make the model achieve highest accuracy and lowest complexity.

Local Optimization Algorithm for LGP:

Step1: Identify basic block

Step2: Go through each basic block with the optimization methods

```
optimization():
        identify basic blocks();
        for each basic block:
        local_optimizations();
        # global_optimizations();

local_optimizations():
        Algebraic Simplification();
        Constant Folding & Flow of Control Optimizations();
        Single Assignment Form();
        Common Subexpression Elimination();
        Copy Propagation();
                (including Copy Propagation and Constant Folding
                Copy Propagation and Dead Code Elimination)
```

Assume the type of the input and output is string.

“r” represents the output instruction below

Pseudo Code:

Step 1: Identify Block:

no labels (except at the first instruction)

no jumps (except in the last instruction)

```
block = {}
block_set = {}
for each instruction:
    if instruction is label or goto:
        if block is not empty: block_set.add(block)
        block.empty()
    block.add(instruction)
```

Step 2: Local Optimization

1. Algebraic Simplification (Delete or Simplify algebraic operations)

```
# Cases when the instructions can be deleted

# Case1: r := x + 0 => delete
if +0 in instruction:
    delete instruction

# Case2: r := x*1 => delete
if *1 in instruction:
    delete instruction

# Cases when the instructions can be algebraic simplified

# Case1: r := x*0 => x:=0
if *0 in instruction:
    output: r = 0

# Case2: transfer power to multiplication
# Example: r := x**2 => r = x*x (**.next = 2 , **.before = x)
if ** in instruction and **.next = 2:
    r = (**.before)   *   (**.before)

# Case3 : transfer multiplication to shift operation
# Example: r := x*8 => r := x<<3 (*.before = x, *.next = 8)
# Example: r := x*15 => t := x<<4; r := t - x
shift_number = *.next // 2

if *.next mod 2 != 0:                          # eg. *next = 15, 15 // 2 = 3
    diff = *.next - 2**n                       # difference = 15 - 2**3 = 7
        output: add t := *.before << shift_number;
                    r := t + diff
else:                                          #*.next mod 2 != 0
    output: r = x << shift_number              # when the base is 2
```

2. Constant Folding & Flow of Control Optimization

operator_list = [+ , - , * , / , **, ...] Page 17

Arithmetic operations: { +, -, ×, / }

Exponential functions: { **, ln(r), e(r), square(r), root(r)}

Trigonomic functions: {sin(r), cos(r)}

Boolean operations: { ∧ ,∨ }

Conditional branches: { if (rj > rk), if (rj ≤ rk), if (rj) }

```
# Example: ri := rj op rk where op.before = rj, op.next = rk (op represents operator)
```

# Case 1: compute operations on constants at compile time
if op.before is numeric and op.next is numeric:
    output: r = op.before op op.next

# Case 2.1: eliminate the instruction and operations if the condition is always false
if ("if" in instruction) and (op.before is numeric & op.next is numeric)
and (op.before op op.next) == false:
    remove the instruction
    remove following operations under the condition

Case 2.2: eliminating unreachable code: how to determine if the block can be reached or not
Step1:
create set_labels {} and set_jumps {}
Step2:
for label in set_labels:
    if label not in set_jumps:
        delete label.block

3. Single Assignment Form (LHS occurs once only)
put all instructions into inst_list[]   # inst_list represents instruction_list
3.1 make LHS register occur once only
step1:
Traverse list and count number of each element. Once occur, count number +1
eg: left_list = [a,b,c,d,a,e,a,a,f,b] => [1,1,1,1,2,1,3,4,1,2]
define register_num: find all 2s in the left_list. Here the register_num is 2, a and b respectively
record index of each elements(a and b) in left_list into register_list[];
here register_list[0]= [0,4,6,7] , register_list[1]=[1,9]

step2:
for i in range(register_num):
    for j in range (len(register_list [i])):
        1.change LHS oldName of inst_list [register_list [j]] into newName
        2. from inst_list [register_list [j]] to inst_list [register_list [j+1]],
          change these oldName appeared in RHS to new Name

4. Common Subexpression Elimination
# Example: r1 = x+y; r2 = x+y => r1 = x+y; r2 = r1
existed_instruction_set = {}   # {RHS: LHS_value}
for each instruction:
    if instruction.RHS in existed_expression_list:
        instruction.RHS = existed_instruction_set[instruction.RHS].LHS_value

5. Copy Propagation

```
# if w := x appears in a block, all subsequent uses of w can be replaced with uses of x.
adict = {}
    for all L:                # L = left side var
        if L in adict:
            del adict[L]
        if len(L.R) == 1:
            adict[L] = L.R
        for term in instruction:
            if term in adict:
                term = adict[term]
```

```
# Copy Propagation and Constant Folding();

shift_number = *.next // 2
if *.next mod 2 != 0:                              # eg. *next = 15, 15 // 2 = 3
    diff = *.next - 2**n                           # difference = 15 - 2**3 = 7
        output: add t := *.before << shift_number;
                r := t + diff
else:                                              #*.next mod 2 != 0
    output: r = x << shift_number                  # when the base is 2

if op.before is numeric and op.next is numeric:
    output: r = op.before op op.next
```

```
# Copy Propagation and Dead Code Elimination();
Collect all instruction in instruction_list = []   # eg: [x := RHS, y := RHS, z := RHS, ... ]
flaglist = [0,0,0, …]                              # flaglist.size() = instruction_list.size()
for i in range len(instruction_list):              # find LHS that appears nowhere only once
    for j in range len(instruction_list):
        if instruction_list [i].LHS   = instruction_list [j].LHS:
            flaglist[i] +=1

# find the index of all elem in flaglist which is not 0, eg: [0,2,0,6,2,0] => [1,3,4]
delete_index = []
for i in len(flaglist):
    if flaglist[i] > 0:
        delete_index.append(i)

for i in delete_index:
    delete instruction_list [index]
```