Final Presenation

Team 4 : Alena Kazakova, 정재환, 김도형

4190.570 Advanced Compiler Construction

Project Objectives

Control Flow Analysis

- TAC to CFG conversion
- Inter-procedural CFG extension

Inter-procedural Analysis

- Context-insensitive
- Handle function parameters and return values

Dataflow Analysis

- Reaching-definition analysis of CFG per process
- Liveness analysis of CFG per process

Constant Propagation

- Based on reaching-definition analysis, do constant propagation
- Modify CFG and TAC accordingly

Dead Code Elimination

- Based on liveness analysis, do dead code elimination
- Modify CFG and TAC accordingly

Control Flow Analysis

- IR: module -> scope -> code block -> instruction
- CFG: graph(module) -> scope graph(code block) -> node(instruction)
- Node has *out-edges* that lead to successor nodes,
  and *in-edges* that come from predecessor nodes.
- For conditional branching instructions, node has 2 out-edges:
  fall-through & jump.
- For function call instruction, node has out-edge to the beginning of
  function's scope, and in-edge from the end of function's scope.

Storing use and def variables

Dataflow analysis:

- Easy for scalar types
- Hard for arrays and pointers
- Hard for branching and function calls

operation	use	def
binary: dst = src1 op src2	src1, src2	dst
unary: dst = op src1	src1	dst
memory: assign dst = src1	src1	dst
conditional branching: if src1 relOp src2 then goto dst	src1,src2	null
unconditional branching: goto dst	null	null
call: dst = call src1	src1	dst
return: return optional src1	src1	null
parameter: dst = index, src1 = parameter	src1	null
reference: dst = &src1	src1	dst
dereference: dst = *src1	src1	dst
type cast: dst = (type)src1	src1	dst
special: jump label and nop	null	null

Reaching-definition Analysis

- Dataflow equation (forward analysis)

$$in[n] = \bigcup_{p \in preds[n]} out[s]$$

$$out[n] = gen[n] \cup (in[n] - kill[n])$$

gen[*n*] – node *n* that defines a variable,
kill[*n*] – set of nodes that define the same variable,
in[*n*] – nodes that reach the beginning of node *n*,
out[*n*] – nodes that reach the end of node *n*

Constant Propagation

- Suppose we have:

  Statement $s_1 : t \leftarrow c$, where c is const

  Statement $s_2 : y \leftarrow x \oplus t$, that uses t
- We know that t is constant in s_2 if s_1 reaches s_2,
- and no other definitions of t reach s_2.
- In this case, we can rewrite s_2 as $y \leftarrow x \oplus c$.

```
Input: Reaching-definition for each CFG
Algorithm:
for each node n in CFG do
        for each src in use[n] do
                if (src type != const)
                        if (src is in reachIn[n])
                                src type := const
                        endif
                endif
        endfor

        if (src1 == const)
                if (op == unary)
                        result := op src1
                        src1 := result
                endif
                if (src2 == const && op == binary)
                        result := src1 op src2
                        src2 := null
                        src1 := result
                endif
                op type := assign
        endif
endfor
```

Liveness Analysis

- Dataflow equation (backward analysis)

$$in[n] = use[n] \cup (out[n] - def[n])$$

$$out[n] = \bigcup_{s \in succs[n]} in[s]$$

use[*n*] – variables used by node *n*,
def[*n*] – variables defined by node *n*,
in[*n*] – nodes that reach the beginning of node *n*,
out[*n*] – nodes that reach the end of node *n*

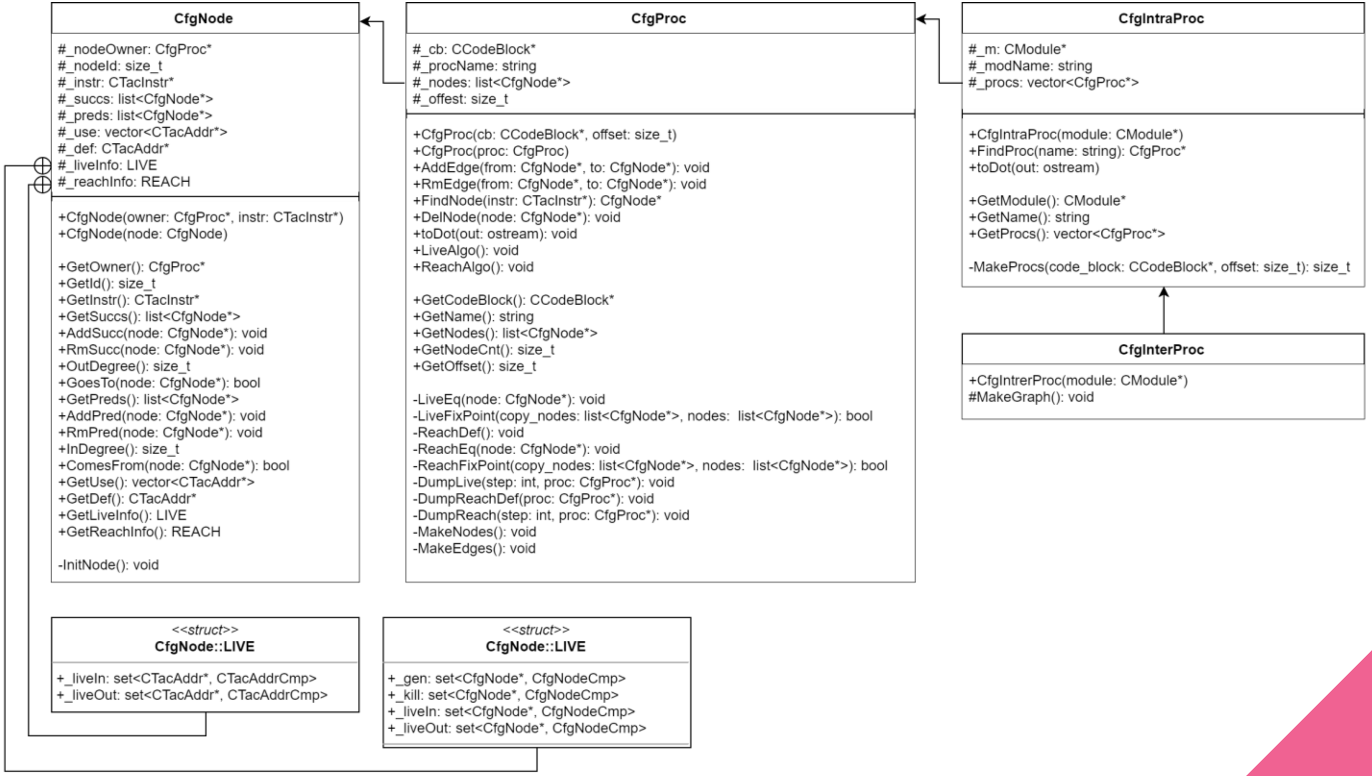
Dead-code Elimination

- Suppose we have:

  Statement $s: a \leftarrow b \oplus c$, such that *a* is not *live-out* of *s*
- Then the statement can be deleted.

```
Input: liveness analysis for each CFG node
Algorithm:
for each node in CFG do
      if (def[n] is in LiveOut[n])
            DeleteNode(n)
      endif
endfor
```

CFG Class Diagram



Demo

`snuplc -tac -const -deadc test/test01.mod`

- Output of reaching-definition analysis and liveness analysis
- TAC & CFG comparison (original vs after constant propagation vs after dead code elimination)

`snuplc -tac test/factorial.mod`

- Intra-procedural CFG vs inter-procedural CFG

Experimental Results (for test01.mod)

Before optimization

- Contains 15 instructions
- test01.mod.s size is 3.6kB

After dead-code elimination

- test01.mod.s size is 3.4kB

After constant propagation

- Contains 6 instructions
- test01.mod.s size is 2.5kB

Time for all cases is 2 msec

Ongoing Work

- Special cases for arrays and pointers
- Inter-procedural analysis
- Testing loops
- Testing recursions