COS320 - Optimization

- Optimization passes: Sequence of IR-IR transformations.
  Each transformation is expected to
    - Improve performance
    - Not change the high-level behavior of program

- Each optimization pass does sth small & simple
  - Combination of passes can yield sophisticated transformations

- Optimization simplifies compiler writing
  - More modular: Can translate into IR in simple but inefficient way, then optimize.

- Optimization simplifies programming
  (Can write somewhat inefficient code).

Whirlwind overview of optimization pass

ex1 Algebraic simplification:

Replace complex expressions with simpler ones
(e.g. pattern match on AST,
e*1 = e, e+0 = e, etc.)
x*4 = x<<2

ex2 loop unrolling: unrolling a loop 4 times

```
i64 array-sum(i64 *a, i64 n)
{ i64 i; i64 sum = 0;
  for (i=0; i<n; ++i) {
      sum += *(a+i); }
  return sum;}
```

```
i64 array-sum(i64 *a, i64 n)
{ i64 i; i64 sum=0;
  for(i=0; i<n%4; ++i){
    sum+=*(a+i); }
  for(; i<n; i+=4) {
     4x
  }
}
```

ex3 strength reduction
(replace expensive operation (e.g. multiplication) w/ cheaper ones (e.g. addition).

compute (i+1, i+1) from (i, i)

row-major →

```
i64 Tr(i64 *m, i64 n){
 i64 i, result=0, *next=m;
 for(i=0; i<n; ++i) {
   result += next;
   next += n+1;
 }
```

(an inductive 'memorization' technique).

Optimization and Program analysis

Conservatively approximate the run-time behavior of a program at compile-time.

- Type inference: find the type of value each expression will evaluate to at run time. Conservative in the sense that analysis will abort if it cannot find a type for a variable, even if one exists.

- Constant propagation: if a variable only holds one a value at runtime, find that value. Conservative in the sense that analysis may fail to find constant values for variables that have them.

basically a form of substitution.

Optimization informed by analysis

- Analysis lets us know which transformations are safe.

- Conservative analysis → Never perform an unsafe optimization, but may miss some safe opt opportunities.

Analyser takes in a Control-flow graph.

- CFG for procedure P is directed rooted graph
  - $G = (N, E, r)$

    Nodes, entry, return nodes.

Constant propagation

for each instruction I:

- A constant environment is a symbol table x to one of
  - int n (x's value is n whenever program is at I)
  - ⊤ (x might take more than one value at I)
  - ⊥ : Unreachable to x
- Motivation: Compute expressions at compile time to save on runtime.

Init: $\{x \mapsto \top, y \mapsto \top, z \mapsto \top\}$

x = add 1, 2 ← Perform static addition.

Next: $\{x \mapsto 3, y \to \top, z \mapsto \top\}$

y = mul x, 11

Next Next: $\{x \to 3, y \to 33, z \to \top\}$

z = add x, y = 3+33 = 36.

Finally: $\{x \to 3, y \to 33, z \to 36\}$.

Propagation of constants thru instructions.

Goal: Constant environment C and instruction

- $x = \text{add}, \text{opn}_1, \text{opn}_2$
- $x = \text{mul}, \text{opn}_1, \text{opn}_2$
- $x = \text{opn}$

Q: Assume const env C holds before instr. What is C' after instr.?

- evaluator for operands:

$$\text{eval}(\text{opn}, C) = \begin{cases} C(\text{opn}) & \text{if opn is available} \\ \text{opn} & \text{if opn is an int.} \end{cases}$$

- Evaluator for instructions.

$$\text{Post}(\text{instr}, C) = \begin{cases} \bot & \text{if } C \text{ is } \bot \\ C\{x \to \text{eval}(\text{opn}, C)\} & \text{if instr is } x = \text{opn} \\ C\{x \mapsto \top\} & \\ C\{x \to \text{eval}(\text{opn}_1, C) + \text{eval}(\text{opn}_2, C)\} & \text{if instr is } x = \text{add opn}_1, \text{opn}_2 \\ C\{x \to \text{eval}(\text{opn}_1, C) \times \text{eval}(\text{opn}_2, C)\} & \text{if instr is } x = \text{mul opn}_1, \text{opn}_2 \end{cases}$$

Propagate constants through basic blocks

- How to propagate const env thru B.B.?
  - Just propagate the last const. env. in B.B.

Across edges (e.g. "branches")

if a block has exactly one predecessor:

☆ Const. env. at entry is constant environment at exit of predecessor.

☆ If a block has multiple predecessors, must combine const environment of both.

↓

Use "merge" operator (Join)

$$\left[\begin{array}{l} e \sqcup \bot = \bot \sqcup e = e \\ (e_1 \sqcup e_2)(x) = \begin{cases} e_1(x) & \text{if } e_1(x) = e_2(x) \\ \top & \text{otherwise} \end{cases} \end{array}\right.$$

Propagating constants through CFG

- Acyclic graphs:

  topsort basic blocks

  propagate const environments forward

  Constant environment for entry node maps each variable to $\top$.

- What about loops?

  Can't topsort because of cyclic dependencies.

take a step back... how to verify const env is correct?

recall a partial order $\sqsubseteq$ is a binary relation that is

- reflexive $a \sqsubseteq a$
- transitive $a \sqsubseteq b \sqsubseteq c \rightarrow a \sqsubseteq c$

Solution: place order on $\mathbb{Z} \cup \{\bot, \top\}$:

$\boxed{\bot \sqsubseteq \quad n \sqsubseteq \top}$ for $n \in \mathbb{Z}$

i.e. (most info to least info).

Now, lift this ordering to const. envs

$\underset{\text{better}}{f} \sqsubseteq g$ iff $f(x) \sqsubseteq g(x)$ $\forall x$.

Smaller is more information, better.

f sends x to $\top \longrightarrow g$ sends x to $\top$

(but converse isn't true).

Now, merge operation $\sqcup$ is least upper bound in this order.

- $t_1 \sqsubseteq (t_1 \sqcup t_2)$ and $t_2 \sqsubseteq (t_1 \sqcup t_2)$
- For any type t' such that
  - $t_1 \sqsubseteq t'$
  - $t_2 \sqsubseteq t'$
- We have $(t_1 \sqcup t_2) \sqsubseteq t'$.

Constant propagation as constraint system

- Let $G = (N, E, s)$ be a CFG.
- For each $bb \in N$, associate two const env's
  - $IN[bb]$ — const env at entry of bb
  - $OUT[bb]$ — const env at exit of bb.
- Say that assignment IN, OUT is conservative if
  - $IN(s)$ assigns each variable $\top$.
  - Foreach $bb \in V$
    $OUT[bb] \sqsupseteq post(bb, IN[bb])$.
  - For each edge $src \to dst \in E$,
    $IN[dst] \sqsupseteq OUT[src]$.

fact: if IN, OUT is conservative, then

- if IN[bb](x) = n
  whenever program execution reaches bb entry the value of x is n.
- if OUT[bb](x) = ⊥, then program execution cannot reach bb.
- Similarly for OUT.

Computing IN, OUT (we want the least solution)

Payoff: when constProp sends x to const (not ⊤) it's better than ⊤.

More const assignments → more opt.

least conservative assignment:

1. IN, OUT conservative
2. if IN', OUT' is conservative, then for any bb we have
   - IN[bb] ⊑ IN'[bb]
   - OUT[bb] ⊑ OUT'[bb]

Computing the least conservative assignment of constant environments

- Initialize IN(S) to the const. environment that sends every variable to $\top$ and OUT(S) to const env that sends every variable to $\bot$.

- Initialize IN[bb] and OUT[bb] to const. env. that sends every variable to $\bot$ for every other basic block.

- Choose a constraint that isn't satisfied by IN, OUT
  - if $\exists$ basic block 'bb' w/ OUT[bb] $\neq$ post(bb, IN[bb]) then set OUT[bb] := post(bb, IN(bb))
  - if $\exists$ edge src $\rightarrow$ dst $\in E$ with IN[dst] $\neq$ OUT[src], then set IN[dst] := IN[dst] $\sqcup$ OUT[src].

- Terminate when all constraints are satisfied.

Properties:
- Algorithm terminates when all constraints are satisfied.