Implementing intraprocedural (global) optimizations

Construct convenient **representation** of procedure body

Control flow graph (**CFG**) captures flow of control

- nodes are IL statements, or whole basic blocks
- edges represent control flow
- node with multiple successors = branch/switch
- node with multiple predecessors = merge
- loop in graph = loop

Data flow graph (**DFG**) capture flow of data

E.g. **def/use chains**:

- nodes are def(inition)s and uses
- edge from def to use
- a def can reach multiple uses
- a use can have multiple reaching defs

Example program

```
x = 3;
y = x * x;
if (y > 10) {
   x = 5;
   y = y + 1;
} else {
   x = 6;
   y = x + 4;
}
w = y / 3;
while (y > 0) {
   z = w * w;
   x = x - z;
   y = y - 1;
}
System.out.println(x);
```

Analysis and transformation

Each optimization is made up of
some number of **analyses**
followed by a **transformation**

Analyze CFG and/or DFG by propagating info
forward or backward along CFG and/or DFG edges

- edges called **program points**
- merges in graph require combining info
- loops in graph require **iterative approximation**

Perform improving transformations based on info computed

- have to wait until any iterative approximation has converged

Analysis must be **conservative**/**safe**/**sound**
so that transformations preserve program behavior

Example: constant propagation & folding

Can use either the CFG or the DFG

CFG analysis info:
table mapping each variable in scope to one of

- a particular constant
- *`NonConstant`*
- *`Undefined`*

Transformation: at each instruction:

- if reference a variable that the table maps to a constant, then replace with that constant (**constant propagation**)
- if r.h.s. expression involves only constants, and has no side-effects, then perform operation at compile-time and replace r.h.s. with constant result (**constant folding**)

For best analysis, do constant folding as part of analysis,
to learn all constants in one pass

Example program

```
x = 3;
y = x * x;
v = y - 2;
if (z > 10) {
   x = 5;
   y = y + 1;
} else {
   y = x + 4;
}
w = y / v;
if (v > 20) {
   v = x - 1;
}
u = x + v;
```

Merging data flow analysis info

How to merge analysis info?

Constraint: merge results must be sound

- if something is believed true after the merge, then it must be true no matter which path we took into the merge
- only things true along all predecessors are true after the merge

To merge two maps of constant info,
build map by merging corresponding variable infos

To merge two variable infos:

- if one is *`Undefined`*, keep the other
- if both same constant, keep that constant
- otherwise, degenerate to *`NonConstant`*

Analysis of loops

How to analyze a loop?

```
i = 0;
x = 10;
y = 20;
while (...) {
   // what's true here?
   ...
   i = i + 1;
   y = 30;
}
// what's true here?
... x ... i ... y ...
```

A safe but imprecise approach:

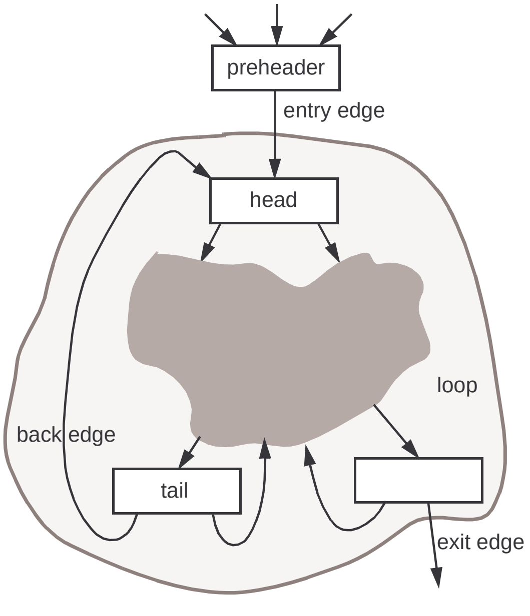
- forget everything when we enter or exit a loop

A precise but unsafe approach:

- keep everything when we enter or exit a loop

Can we do better?

Some loop terminology



Optimistic iterative analysis

1. Assuming info at loop head is same as info at loop entry

2. Then analyze loop body, computing info at back edge

3. Merge infos at loop back edge and loop entry

4. Test if merged info is same as original assumption

   a. If so, then we're done

   b. If not, then replace previous assumption with merged info, and goto step 2

Example

```
i = 0;
x = 10;
y = 20;
while (...) {
   // what's true here?
   ...
   i = i + 1;
   y = 30;
}
// what's true here?
... x ... i ... y ...
```

Why does optimistic iterative analysis work?

Why are the results always conservative?

Because if the algorithm stops, then

- the loop head info is at least as conservative as both the loop entry info and the loop back edge info
- the analysis within the loop body is conservative, given the assumption that the loop head info is conservative

Why does the algorithm terminate?

It might not!

But it does if:

- there are only a finite number of times we could merge values together without reaching the worst case info (e.g. `NotConstant`)

Another example: live variables analysis

Want the set of variables that are **live** at each pt. in program

- live: *might* be used *later* in the program

Supports dead assignment elimination, register allocation

What info computed for each program point?

What is the requirement for this info to be conservative?

How to merge two infos conservatively?

How to analyze an assignment, e.g. `X := Y + Z`?

- given `liveVars` before (or after?), what is computed after (or before?)

What is live at procedure entry (or exit?)?

Example