CSE401: Optimization

David Notkin
Autumn 2000

Interprocedural optimizations

- What happens if we expand the scope of the optimizer to include procedures calling each other
  - In the broadest scope, this is optimization of the program as a whole
- We can do local, intraprocedural optimizations at a bigger scope
  - For example, constant propagation
- But we can also do entirely new optimizations, such as inlining

---

Inlining

- Replace procedure call with the body of the called procedure

```
const pi:real := 3.14159;
proc area(rad:int):int;
begin
  return pi*(rad^2);
end;
…
r := 5;
…
output := area(r);
```

```
const pi:real := 3.14159;
proc area(rad:int):int;
begin
  return pi*(rad^2);
end;
…
r := 5;
…
output := pi*(r^2);
```

Questions about inlining:

few answers

- How to decide where the payoff is sufficient to inline?
  - The real decision depends on dynamic information about frequency of calls
- In most cases, inlining causes the code size to increase; when is this acceptable?
- Others?

Optimization and debugging

- Debugging optimized code is often challenging
- Examples include
  - What if statements are no longer ordered as they were in the source code?
  - What if variables in the source code are eliminated?
  - What if code is inlined?
- In general, the more optimization there is, the more complex the back-mapping is from the target code to the source code … which can confuse a programmer

---

Summary of optimization

- Larger scope of analysis yields better results
  - Most of today's optimizing compilers work at the intraprocedural level, with some doing some work at the interprocedural level
- Optimizations are usually organized as collections of passes
- The presence of optimizations may make other parts of the compiler (e.g., the code generator) easier to write
  - One example was to use a simple instruction selection algorithm, knowing that the optimizer can, in essence, act to improve these instruction selections

Implementing intraprocedural optimizations

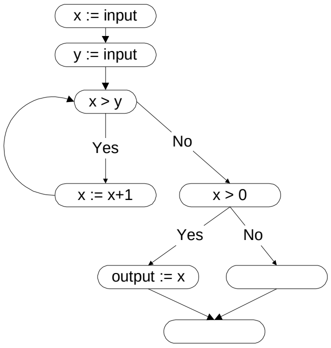
- The heart of implementing optimizations is the definition and construction of a convenient representation
- We'll look a bit more closely at two common and useful representations, which I've mentioned before
  - The control flow graph (CFG)
  - The data flow graph (DFG)

CFG

- Nodes are intermediate language statements
  - Or whole basic blocks
- Edges represent control flow
- Node with multiple successors is a branch/switch
- Node with multiple predecessors is a merge
- Loop in a graph represents a loop in the program

Example

```
while x > y do
  x : = x + 1;
end;
if x > 0 then
  output := x;
end;
```

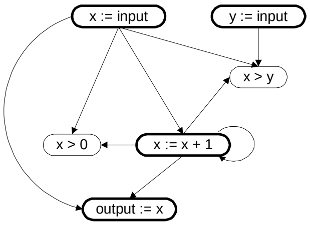


DFG: def/use chains

- Nodes are def(initions) and uses
- Edge from def to use
- A def can reach multiple uses
- A use can have multiple reaching defs

Example

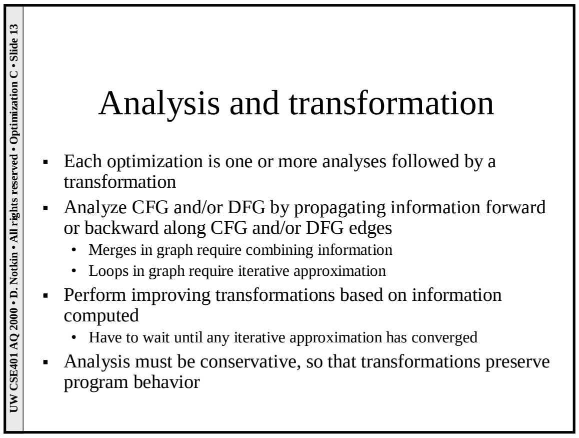
```
x := input;
y := input;
while x > y do
  x : = x + 1;
end;
if x > 0 then
  output := x;
end;
```



Example program

CFG and DFG in groups

```
x := 3;
y := x * x;
if y > 10 then
  x := 5;
  y := y + 1;
else
  x := 6;
  y := x + 4;
end;
w := y / 3;
while y > 0 do
  z := w * w;
  x := x - z;
  y := y - 1;
end;
output  := x;
```



Analysis and transformation

- Each optimization is one or more analyses followed by a transformation
- Analyze CFG and/or DFG by propagating information forward or backward along CFG and/or DFG edges
  - Merges in graph require combining information
  - Loops in graph require iterative approximation
- Perform improving transformations based on information computed
  - Have to wait until any iterative approximation has converged
- Analysis must be conservative, so that transformations preserve program behavior