Today

- Tour of optimizations
- Dataflow analysis intro

Example 1

```
int foo (int z) {
   x = 3 + 6;
   y = x - 5;
   return z * y;
}
```

- What optimizations apply?

Example 1

```
int foo (int z) {
   x = 3 + 6;         => x = 9; (constant folding)
   y = x - 5;
   return z * y;
}
```

- What optimizations apply?

Example 1

```
int foo (int z) {
   x = 3 + 6;         => x = 9; (constant folding)
   y = x - 5;         => 9 - 5 (const prop)
   return z * y;
}
```

- What optimizations apply?

Example 1

```
int foo (int z) {
   x = 3 + 6;         => x = 9; (constant folding)
   y = x - 5;         => 9 - 5 (const prop) => 4 (CF)
   return z * y;
}
```

- What optimizations apply?

Example 1

```
int foo (int z) {
   x = 3 + 6;         => x = 9; (constant folding)
   y = x - 5;         => 9 - 5 (const prop) => 4 (CF)
   return z * y;      => z * 4 (CP)
}
```

- What optimizations apply?

Example 1

```
int foo (int z) {
   x = 3 + 6;          => x = 9; (constant folding)
   y = x - 5;          => 9 - 5 (const prop) => 4 (CF)
   return z * y;       => z * 4 (CP) => z << 2
}                                (strength reduction)
```

- **What optimizations apply?**

Example 2

```
x = a + b;
   ...
y = a + b;
```

- **In what circumstances can (a+b) by replaced by x?**
- **This is common subexpression elimination (CSE) – replacing expressions that evaluate to the same value with a single variable holding the computed value**

Example 3

```
if (...) {
   x = a + b;
}

...

y = a + b;
```

- **Can we change the assignment to read** `y = x;` **?**

Example 3

- **Can we change the assignment to read** `y = x;` **?**
- **No, but we can apply partial redundancy elimination (PRE), a generalization of common subexpression elimination:**

```
if (...) {
   t = a + b;
   x = t;
} else {
   t = a + b;
}

...

y := t;
```

Example 4

```
x = y;
...
z = z + x;
```

- **When can we replace x with y in the second assignment?**
- **This is copy propagation – replacing the occurrences of targets of direct assignments with their values**

Example 4

```
x = y**z;
...
x = 15;
```

- **When can the first assignment be removed?**
- **This is dead assignment elimination (DAE)**
  - **Often cleans up after other optimizations**

```
x = y          Copy prop    x = y          DAE
z = z + x        ==>        z = z + y      ==>    z = z + y
```

Example 5

```
if (false) {
  ...
}
```

- When can this statement be removed?
- Dead code elimination (DCE) – another cleanup

Example 6

- Java code:

```
a = new int [10];
for (index = 0; index < 10; index ++) {
   a[index] = 100;
}
```

Example 6

- “Lowered” Java code:

```
a = new int [10];
for (index = 0; index < 10; index ++) {
   if (index < 0 || index >= a.length()) {
      throw OutOfBoundsException;
   }
   a[index] = 100;
}
```

Example 6

- “Lowered” Java code:

```
a = new int [10];
for (index = 0; index < 10; index ++) {
   if (index < 0 || index >= 10) {
      throw OutOfBoundsException;
   }
   a[index] = 100;
}
```

Example 6

- “Lowered” Java code:

index $\in$ [0..9]

```
a = new int [10];
for (index = 0; index < 10; index ++) {
   if (index < 0 || index >= 10) {
      throw OutOfBoundsException;
   }
   a[index] = 100;
}
```

Example 6

- “Lowered” Java code:

```
a = new int [10];
for (index = 0; index < 10; index ++) {
   if (false) {
      throw OutOfBoundsException;
   }
   a[index] = 100;
}
```

Example 6

- "Lowered" Java code:

```
a = new int [10];
for (index = 0; index < 10; index ++) {
  a[index] = 100;
}
```

Example 7

```
p = &x;
*p = 5;
y = x + 1;
```

- Want to turn this into:

```
p = &x;
x = 5;
y = 6;
```

- This is must alias analysis – *p must refer to x

Example 8

```
for j = 1 to N
   for i = 1 to N
      a[i] = a[i] + b[j];
```

- Want to turn this into:

```
for j = 1 to N
   t = b[j];
   for i = 1 to N
      a[i] = a[i] + t;
```

- This is loop invariant code motion (LICM), or hoisting – executing code fewer times by moving it out of a loop, when this can be done safely

Example 9

```
area (h,w) { return h * w; }

h = ...;
w = 4;
a = area (h,w)
```

- Want to turn this into:

```
h = ...;
a = h << 2;
```

- Requires: inlining + copy prop + DAE + strength reduction

Optimization Themes

- Don't compute if you don't have to
  - Unused assignment elimination, DCE
- Compute at compile-time if possible
  - Constant folding, loop unrolling, inlining
- Compute it as few times as possible
  - CSE, PRE, loop invariant code motion
- Compute it as cheaply as possible
  - Strength reduction, inlining
- Enable other optimizations
  - Constant and copy prop, inlining, pointer analysis

Overall

- Collection of relatively simple optimizations, working together and run repeatedly, can substantially improve code
- Questions
  - What order should optimizations be run in?
  - How many times should they be run?
  - Should the compiler try to optimize "stupid" code?
  - How to decide whether to apply optimizations that do not uniformly speed up code?

Dataflow analysis: What is it?

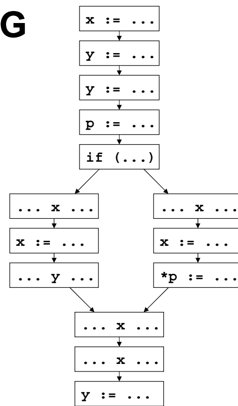
- A common framework for expressing algorithms that compute facts about programs
  - Facts must be true across all possible executions
  - Approximation is inevitable
- Common framework is useful in order to
  - Reason about correctness of analyses
  - Communicate analyses to others
  - Compare analyses
  - Adapt techniques from one analysis to another
  - Reuse implementations

Control Flow Graphs

- For now, we will use a Control Flow Graph (CFG) representation of programs
  - Each statement becomes a node
  - Edges between nodes represent control flow
- Later we will see other program representations
  - Variations on the CFG (e.g. CFG with basic blocks)
  - Other graph based representations

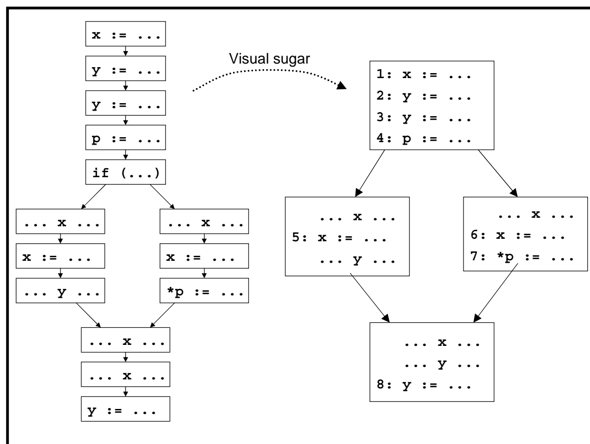
Example CFG

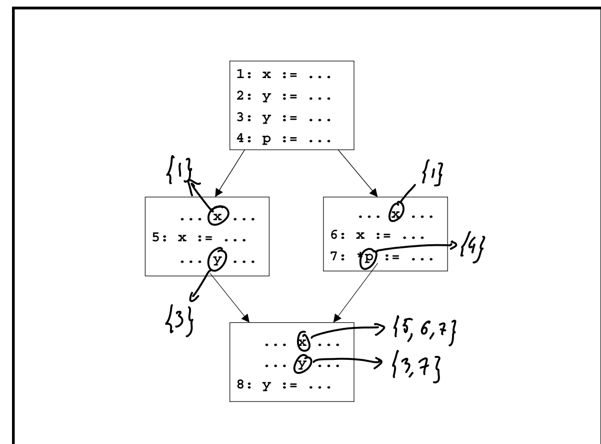
```
x := ...
y := ...
y := ...
p := ...
if (...) {
   ... x ...
   x := ...
   ... y ...
}
else {
   ... x ...
   x := ...
   *p := ...
}
... x ...
... y ...
y := ...
```



Example dataflow analysis: Reaching Definitions

- For each use of a variable, determine what assignments could have set the value being read from the variable
- Useful for
  - Performing constant and copy prop
  - Detecting references to undefined variables
  - Presenting “def/use chains” to the programmer
  - Building other representations, like the DFG
- Let’s try this out on an example





Safety

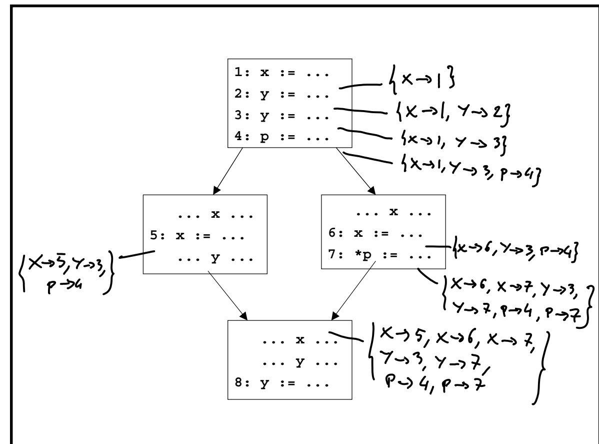
- Recall intended use of this info
  - Performing constant and copy prop
  - Detecting references to undefined variables
  - Presenting "def/use chains" to the programmer
  - Building other representations, like the DFG
- Safety criteria
  - Cannot miss any bindings
    - This would cause subsequent optimizations to change program semantics
  - Can include bindings that do not correspond to possible executions
    - But this may prevent some optimizations from firing

Towards DFA

- What does a DFA framework do?
  - Computes information at each program point (edge) in the CFG
  - So, generalize the reaching definitions problem by stating what should be computed at each program point
- "For each program point in the CFG, compute the set of definitions (statements) that may reach that point"
- Notion of safety remains the same

Reaching Definitions, Generalized

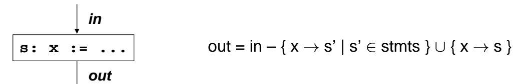
- Computed information at a program point is a set of var → stmt bindings
  - E.g. $\{ x \rightarrow s_1, x \rightarrow s_2, y \rightarrow s_3 \}$
- How do we get the previous info we wanted?
  - If a var *x* is used in a stmt whose incoming info is *in*, then: $\{ s \mid (x \rightarrow s) \in in \}$
- This is a common pattern
  - Generalize the problem to define what information should be computed at each program point
  - Use the computed information at the program points to get the original info we wanted



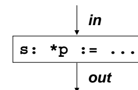
Formalizing DFA

- Let's try to precisely express the algorithms for computing dataflow information
- We'll model DFA as solving a system of constraints
- Each node in the CFG will impose constraints relating information at predecessor and successor points
- Solution to constraints is result of analysis

Constraints for reaching definitions



out = in – { x → s' | s' ∈ stmts } ∪ { x → s }

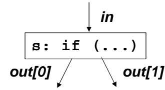


- Using may-point-to information:

  out = in ∪ { x → s | x ∈ may-point-to(p) }
- Using must-point-to as well:

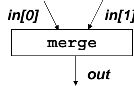
  out = in – { x → s' | x ∈ must-point-to(p) ∧ s' ∈ stmts } ∪ { x → s | x ∈ may-point-to(p) }

Constraints for reaching definitions



out [0] = *in* ∧
out [1] = *in*

more generally: $\forall\, i\,.\; out[\,i\,] = in$



$out = in[\,0\,] \cup in[\,1\,]$

more generally: $out = \bigcup_i in[\,i\,]$

Transfer Functions

- **The constraint for a statement kind s often has the form: out = F_s(in)**
- **F_s is called a transfer function (or flow function, or abstract transfer function)**
  - **Recall the transfer functions we looked at last week, for merge and bitwise-and in the ternary-bit domain**
- **Given information *in* before statement s, F_s(*in*) returns information after statement s**

Transfer Functions

- **Transfer functions are a central component of a dataflow analysis**
  - **They state constraints on the information flowing into and out of a statement**
- **Today we looked at some forward transfer functions**
  - **They can work backwards as well**

Summary

- **Dataflow analysis provides a generic framework for plugging program analyses into**
- **We've just scratched the surface**
- **Questions to think about:**
  - **Where do transfer functions come from?**
  - **Are there generic requirements for correct transfer functions?**
  - **How do we know if transfer functions are correct?**
  - **Is there always a unique solution?**
  - **Is dataflow analysis guaranteed to terminate?**
  - **What algorithms are suited to solving these systems of equations?**
  - **How can the CFG representation be improved to make the analysis faster?**