**Prelude**
- How many hours of sleep did you get?
- What are the stages of sleep?
  - Non-REM 1-3
  - REM
- What is the purpose of sleep?
  - Incompletely understood – multiple functions
  - Restorative state – anabolic (growth hormones, repair mechanisms)
  - Memory consolidation, maintenance
  - Interesting: N3 (slow wave) for declarative memory, REM for procedural and spatial memory
  - “Random” firing to “unlearn” weak memories, renew network
- What is the record for going without sleep?
  - 264 hours (Randy Gardner)

**Last time**
- Lowering
  - From language-level constructs to machine-level constructs
- At this point we could generate machine code
  - Output of lowering is a correct translation
  - What’s left to do?
    - Map from lower-level IR to actual ISA
    - Maybe some register management
    - (could be required)
  - Why have a separate assembler?
    - Handles “packing the bits”
    - Addi <target>, <source>, <value>

**Optimizations**
- What are they?
  - Code transformations
  - Improve some metric
- Metrics
  - Performance: time, instructions, cycles
  - Are these metrics equivalent?
  - Memory
    - Memory hierarchy (reduce cache misses)
    - Reduce memory usage
  - Code Size
  - Energy

**But first…**
- The compiler “understands” the program
  - IR captures program semantics
  - Lowering: semantics-preserving transformation
  - Why not do others?
- Compiler optimizations
  - Oh great, now my program will be optimal!
  - Sorry, it’s a misnomer
  - What is an “optimization”?

**Big picture**
Why optimize?
- High-level constructs may make some optimizations difficult or impossible:
  \[ A[i][j] = A[i][j-1] + 1 \]
  \[ t = A + i \times \text{row} + j \]
  \[ s = A + i \times \text{row} + j - 1 \]
  \[ (*t) = (*s) + 1 \]
- High-level code may be more desirable
- Program at high level
- Focus on design, clean, modular implementation
- Let compiler worry about gory details
- Premature optimization is the root of all evil!

Why optimize?
- Program at high level
- Focus on design, clean, modular implementation
- Let compiler worry about gory details
- Premature optimization is the root of all evil!

Limitations
- What are optimizers good at?
  - Consistent and thorough
  - Find all opportunities for an optimization
  - Uniformly apply the transformation
- What are they not good at?
  - Asymptotic complexity
  - Compilers can’t fix bad algorithms
  - Compilers can’t fix bad data structures
- There’s no magic

Requirements
- Safety
  - Preserve the semantics of the program
  - What does that mean?
- Profitability
  - Will it help our metric?
  - Do we need a guarantee of improvement?
- Risk
  - How will it interact with other optimizations?
  - How will it affect other stages of compilation?

Example: loop unrolling
- Safety:
  - Always safe; getting loop conditions right can be tricky.
- Profitability
  - Depends on hardware – usually a win – why?
- Risk
  - Increases size of code in loop
  - May not fit in the instruction cache

Optimizations
- Many, many optimizations invented
  - Constant folding, constant propagation, tail-call elimination, redundancy elimination, dead code elimination, loop-invariant code motion, loop splitting, loop fusion, strength reduction, array scalarization, inlining, cloning, data prefetching, parallelization, etc.
- How do they interact?
  - Optimist: we get the sum of all improvements!
  - Realist: many are in direct opposition
Rough categories

- Traditional optimizations
  - Transform the program to reduce work
  - Don’t change the level of abstraction
- Resource allocation
  - Map program to specific hardware properties
  - Register allocation
  - Instruction scheduling, parallelism
  - Data streaming, prefetching
- Enabling transformations
  - Don’t necessarily improve code on their own
    - Inlining, loop unrolling

Constant propagation

- Idea
  - If the value of a variable is known to be a constant at compile-time, replace the use of variable with constant

\[
\begin{align*}
  n &= 10; \\
  c &= 2; \\
  \text{for } (i=0; i<10; i++) \\
  s &= s + i^2; \\
  \end{align*}
\]

- Safety
  - Prove the value is constant
- Notice
  - May interact favorably with other optimizations, like loop unrolling – now we know the trip count

Constant folding

- Idea
  - If operands are known at compile-time, evaluate expression at compile-time

\[
\begin{align*}
  r &= 3.141 \times 10; \\
  r &= 31.41; \\
  x &= A[2]; \\
  t1 &= 2 \times t; \\
  t2 &= A + t1; \\
  x &= t2; \\
  \end{align*}
\]

- What do we need to be careful of?
  - Is the result the same as if executed at runtime?
  - Overflow/underflow, rounding and numeric stability
- Often repeated throughout compiler

Partial evaluation

- Constant propagation and folding together
- Idea:
  - Evaluate as much of the program at compile-time as possible
  - More sophisticated schemes:
    - Simulate data structures, arrays
    - Symbolic execution of the code
- Caveat: floating point
  - Preserving the error characteristics of floating point values

Algebraic simplification

- Idea:
  - Apply the usual algebraic rules to simplify expressions

\[
\begin{align*}
  a \cdot 1 &= a \\
  a/1 &= a \\
  a + 0 &= a \\
  b \lor \text{false} &= b \\
  \end{align*}
\]

- Repeatedly apply to complex expressions
- Many, many possible rules
  - Associativity and commutativity come into play
  - What could we do to the code to help?

Common sub-expression elimination

- Idea:
  - If program computes the same expression multiple times, reuse the value.

\[
\begin{align*}
  a &= b + c; \\
  c &= b + c; \\
  d &= b + c; \\
  \end{align*}
\]

- Safety
  - Subexpression can only be reused until operands are redefined
- Often occurs in address computations
  - Array indexing and struct/field accesses
Dead code elimination

- **Idea:**
  - If the result of a computation is never used, then we can remove the computation
  
  
  \[ \begin{align*}
  x &= y + 1; \\
  y &= 1; \\
  x &= 2 \times z;
  \end{align*} \]

- **Safety:**
  - Variable is dead if it is never used after defined
  - Remove code that assigns to dead variables
  - This may, in turn, create more dead code
  - Dead-code elimination usually works transitively

Copy propagation

- **Idea:**
  - After an assignment \( x = y \), replace any uses of \( x \) with \( y \)
  
  \[ \begin{align*}
  x &= y; \\
  \text{if (} x = 1 \text{)} s &= x + f(x); \\
  y &= 1; \\
  s &= y + f(y);
  \end{align*} \]

- **Safety:**
  - Only apply up to another assignment to \( x \), or
  - …another assignment to \( y \)!
  - What if there was an assignment \( y = z \) earlier?
  - Apply transitively to all assignments

Unreachable code elimination

- **Idea:**
  - Eliminate code that can never be executed
  
  ```
  #define DEBUG 0
  . . .
  if (DEBUG)
    print("Current value = ", v);
  ```

- **Different implementations**
  - High-level: look for if (false) or while (false)
  - Low-level: more difficult
  - Code is just labels and gotos
  - Traverse the graph, marking reachable blocks

How do these things happen?

- **Who would write code with:**
  - Dead code
  - Common subexpressions
  - Constant expressions
  - Copies of variables

- **Two ways they occur**
  - High-level constructs – we’ve already seen examples
  - Other optimizations
    - Copy propagation often leaves dead code
    - Enabling transformations; inlining, loop unrolling, etc.

Loop optimizations

- **Program hot-spots are usually in loops**
  - Most programs: 90% of execution time is in loops
  - What are possible exceptions?
    - OS kernels, compilers and interpreters

- **Loops are a good place to expend extra effort**
  - Numerous loop optimizations
  - Often expensive – complex analysis
  - For languages like Fortran, very effective
  - What about C?

Loop-invariant code motion

- **Idea:**
  - If a computation won’t change from one loop iteration to the next, move it outside the loop
  
  \[ \begin{align*}
  \text{for } i=0; i<N; i++ \\
  A[i] &= A[i] + x^2; \\
  t1 &= x^2; \quad \text{for } i=0; i<N; i++ \\
  A[i] &= A[i] + t1;
  \end{align*} \]

- **Safety:**
  - Determine when expressions are invariant
  - Just check for variables with no assignments?
  - Useful for array address computations
  - Not visible at source level
Strength reduction

- **Idea:**
  - Replace expensive operations (multiplication, division) with cheaper ones (addition, subtraction, bit shift)
- Traditionally applied to **induction variables**
  - Variables whose value depends linearly on loop count
  - Special analysis to find such variables

```c
for (i=0;i<N;i++)
  v = 4*i;
```

```c
v = 0;
for (i=0;i<N;i++)
  v = v + 4;
```

// Typical example of premature optimization

- Programmers use bit-shift instead of multiplication
- “x<<2” is harder to understand
- Most compilers will get it right automatically

```c
x * 2
x * 2^c
x/2^c
x + x
x<<c
x>>c
```

Inlining

- **Idea:**
  - Replace a function call with the body of the callee
- **Safety**
  - What about recursion?
- **Risk**
  - Code size
  - Most compilers use heuristics to decide when
  - Has been cast as a knapsack problem

```c
class A { void M() {…} }
class B extends A { void M() {…} }
void foo(A x)
{   // which M?
  x.M();
}
```

Inlining in Java

- **With guards:**

```java
void foo(A x)
{   if (x is type A)
    x.M(); // inline A’s M
    if (x is type B)
      x.M(); // inline B’s M
}
```

- **Specialization**
  - At a given call, we may be able to determine the type
  - Requires fancy analysis
Big picture

- When do we apply these optimizations?
  - High-level:
    - Inlining, cloning
    - Some algebraic simplifications
  - Low-level
    - Everything else
- It’s a black art
  - Ordering is often arbitrary
  - Many compilers just repeat the optimization passes over and over

Writing fast programs

In practice:

- Pick the right algorithms and data structures
- Asymptotic complexity and constants
- Memory usage, memory layout, data representation
- Turn on optimization and profile
- Run-time
  - Program counters (e.g., cache misses)
- Evaluate problems
- Tweak source code
  - Help the optimizer do “the right thing”

Anatomy of an optimization

Two big parts:

- Program analysis
  - Pass over code to find:
    - Opportunities
    - Check safety constraints
- Program transformation
  - Change the code to exploit opportunity
- Often: rinse and repeat

Dead code elimination

- Idea:
  - Remove a computation if result is never used

  \[
  \begin{align*}
  y &= w - 7; \\
  x &= y + 1; \\
  y &= 1; \\
  x &= 2 \times z; \\
  y &= w - 7; \\
  y &= 1; \\
  x &= 2 \times z; \\
  y &= 1; \\
  x &= 2 \times z; \\
  \end{align*}
  \]

- Safety
  - Variable is dead if it is never used after defined
  - Remove code that assigns to dead variables
- This may, in turn, create more dead code
- Dead-code elimination usually works transitively

Dead code

- Another example:
  \[
  \begin{align*}
  x &= y + 1; \\
  y &= 2 \times z; \\
  x &= y + z; \\
  z &= 1; \\
  z &= x; \\
  \end{align*}
  \]

- Conditions:
  - Computations whose value is never used
  - Obvious for straight-line code
  - What about control flow?

- With if-then-else:
  \[
  \begin{align*}
  x &= y + 1; \\
  y &= 2 \times z; \\
  \text{if } (c) x &= y + z; \\
  z &= 1; \\
  z &= x; \\
  \end{align*}
  \]

- Which statements are dead code?
  - What if “c” is false?
  - Dead only on some paths through the code
Dead code
- And a loop:

```c
while (p) {
  x = y + 1;
  y = 2 * z;
  if (c) x = y + z;
  z = 1;
  }
  z = x;
```
- Now which statements are dead code?

Low-level IR
- Most optimizations performed in low-level IR
  - Labels and jumps
  - No explicit loops
  - Even harder to see possible paths

```c
label1: jumpifnot p label2
  x = y + 1
  y = 2 * z
jumpifnot c label3
  x = y + z
label3:
  z = 1
  jump label1
label2: z = x
```

Optimizations and control flow
- Dead code is flow sensitive
- Not obvious from program
  - Dead code example: are there any possible paths that make use of the value?
  - Must characterize all possible dynamic behavior
  - Must verify conditions at compile-time
- Control flow makes it hard to extract information
  - High-level: different kinds of control structures
  - Low-level: control-flow hard to infer
- Need a unifying data structure

Control flow graph
- Control flow graph (CFG):
  - A graph representation of the program
  - Includes both computation and control flow
  - Easy to check control flow properties
  - Provides a framework for global optimizations and other compiler passes
- Nodes are basic blocks
  - Consecutive sequences of non-branching statements
- Edges represent control flow
  - From jump to a label
  - Each block may have multiple incoming/outgoing edges

CFG Example
- Program:
  ```c
  x = a + b;
  y = 5;
  if (c) {
    x = x + 1;
    y = y + 1;
  } else {
  }
  z = x + y;
```
- Control flow graph:

```
BB1: x = a + b; y = 5; if (c) {
  x = x + 1;
  y = y + 1;
} else {
}
BB2: z = x + y;
BB3: x = x - 1;
BB4: y = y - 1;
BB5: x = x + 1;
BB6: y = y + 1;
BB7: x = x - 1;
BB8: y = y - 1;
```
Multiple program executions

- CFG models all program executions
- An actual execution is a path through the graph
- Multiple paths: multiple possible executions
  - How many?

Control flow graph

Execution 1

- CFG models all program executions
- Execution 1:
  - c is true
  - Program executes BB₁, BB₂, and BB₄

Basic blocks

- Idea:
  - Once execution enters the sequence, all statements (or instructions) are executed
  - Single-entry, single-exit region
- Details
  - Starts with a label
  - Ends with one or more branches
  - Edges may be labeled with predicates
    - May include special categories of edges
      - Exception jumps
      - Fall-through edges
      - Computed jumps (jump tables)

Building the CFG

- Two passes
  - First, group instructions into basic blocks
  - Second, analyze jumps and labels
- How to identify a basic blocks?
  - Non-branching instructions
    - Control cannot flow out of a basic block without a jump
  - Non-label instruction
    - Control cannot enter the middle of a block without a label

Basic blocks

- Basic block starts:
  - At a label
  - After a jump
- Basic block ends:
  - At a jump
  - Before a label
Basic blocks

- Basic block start:
  - At a label
  - After a jump

- Basic block end:
  - At a jump
  - Before a label

- Note: order still matters

Add edges

- Unconditional jump
  - Add edge from source of jump to the block containing the label

- Conditional jump
  - 2 successors
  - One may be the fall-through block

- Fall-through

Two CFGs

- From the high-level
  - Break down the complex constructs
  - Stop at sequences of non-control-flow statements
  - Requires special handling of break, continue, goto

- From the low-level
  - Start with lowered IR – tuples, or 3-address ops
  - Build up the graph
  - More general algorithm
  - Most compilers use this approach

- Should lead to roughly the same graph

Using the CFG

- Uniform representation for program behavior
  - Shows all possible program behavior
  - Each execution represented as a path
  - Can reason about potential behavior
    - Which paths can happen, which can’t?
  - Possible paths imply possible values of variables

- Example: liveness information

- Idea:
  - Define program points in CFG
  - Describe how information flows between points

Program points

- In between instructions
  - Before each instruction
  - After each instruction

Live variables analysis

- Idea
  - Determine live range of a variable
    - Region of the code between when the variable is assigned and when its value is used
  - Specifically:
    - Def: A variable v is live at point p if
      - There is a path through the CFG from p to a use of v
      - There are no assignments to v along the path
  - Compute a set of live variables at each point p

- Uses of live variables:
  - Dead-code elimination – find unused computations
  - Also: register allocation, garbage collection
Computing live variables
- How do we compute live variables?
  (Specifically, a set of live variables at each program point)
- What is a straightforward algorithm?
  - Start at uses of v, search backward through the CFG
  - Add v to live variable set for each point visited
  - Stop when we hit assignment to v
- Can we do better?
  - Can we compute liveness for all variables at the same time?
    Idea:
    - Maintain a set of live variables
    - Push set through the CFG, updating it at each instruction

Flow of information
- Question 1: how does information flow across instructions?
- Question 2: how does information flow between predecessor and successor blocks?

Live variables analysis
- At each program point:
  Which variables contain values computed earlier and needed later
- For instruction I:
  - \( \text{in}[I] \): live variables at program point before I
  - \( \text{out}[I] \): live variables at program point after I
- For a basic block B:
  - \( \text{in}[B] \): live variables at beginning of B
  - \( \text{out}[B] \): live variables at end of B
- Note: \( \text{in}[I] = \text{in}[B] \) for first instruction of B
  \( \text{out}[I] = \text{out}[B] \) for last instruction of B

Computing liveness
- Answer question 1: for each instruction I, what is relation between \( \text{in}[I] \) and \( \text{out}[I] \)?
- Answer question 2: for each basic block B, with successors \( B_1, \ldots, B_n \), what is relationship between \( \text{out}[B] \) and \( \text{in}[B_1] \ldots \text{in}[B_n] \)

Part 1: Analyze instructions
- Live variables across instructions
- Examples:
  - \( \text{in}[I] = \{y,z\} \)
    \( \text{out}[I] = \{x\} \)
  - \( \text{in}[I] = \{y,z,t\} \)
    \( \text{out}[I] = \{x\} \)
  - \( \text{in}[I] = \{x,t\} \)
    \( \text{out}[I] = \{x,t\} \)
- Is there a general rule?

Liveness across instructions
- How is liveness determined?
  - All variables that I uses are live before I
    Called the uses of I
  - All variables live after I are also live before I, unless I writes to them
    Called the defs of I
  - Mathematically:
    \( \text{in}[I] = (\text{out}[I] - \text{def}[I]) \cup \text{use}[I] \)
Example

- Single basic block
  - \( \text{Live}_1 = \text{in}[B] = \text{in}[I_1] \)
  - \( \text{Live}_2 = \text{out}[I_1] = \text{in}[I_2] \)
  - \( \text{Live}_3 = \text{out}[I_2] = \text{in}[I_3] \)
  - \( \text{Live}_4 = \text{out}[I_3] = \text{out}[B] \)

- Relation between live sets
  - \( \text{Live}_1 = (\text{Live}_2 - \{x\}) \cup \{y\} \)
  - \( \text{Live}_2 = (\text{Live}_3 - \{y\}) \cup \{z\} \)
  - \( \text{Live}_3 = (\text{Live}_4 - \{\}) \cup \{d\} \)

Flow of information

- Equation:
  - \( \text{in}[I] = (\text{out}[I] - \text{def}[I]) \cup \text{use}[I] \)

- Notice: information flows **backwards**
  - Need \( \text{out} \) sets to compute \( \text{in} \) sets
  - Propagate information up

- Many problems are **forward**
  - Common sub-expressions, constant propagation, others

Part 2: Analyze control flow

- **Question 2:** for each basic block \( B \), with successors \( B_1, \ldots, B_n \), what is relationship between \( \text{out}[B] \) and \( \text{in}[B_1] \ldots \text{in}[B_n] \)

- Example:

  - \( \text{B} \)
  - \( \text{out}[] \)
  - \( \text{in}[] = \{\} \)
  - \( \text{w} = \text{x} + \text{z}; \text{B}_1 \)
  - \( \text{q} = \text{x} + \text{y}; \text{B}_n \)

- What’s the general rule?

Control flow

- Rule: A variable is live at end of block \( B \) if it is live at the beginning of **any** of the successors
  - Characterizes all possible executions
  - **Conservative:** some paths may not actually happen

- Mathematically:
  - \( \text{out}[B] = \bigcup_{B' \in \text{succ}(B)} \text{in}[B'] \)

- Again: information flows backwards

System of equations

- Put parts together:
  - \( \text{in}[I] = (\text{out}[I] - \text{def}[I]) \cup \text{use}[I] \)
  - \( \text{out}[B] = \bigcup_{B' \in \text{succ}(B)} \text{in}[B'] \)

- Defines a system of equations (or constraints)
  - Consider equation instances for each instruction and each basic block
  - What happens with loops?
    - Circular dependences in the constraints
    - Is that a problem?

Solving the problem

- Iterative solution:
  - Start with empty sets of live variables
  - Iteratively apply constraints
  - Stop when we reach a **fixpoint**

  - For all instructions \( \text{in}[I] = \text{out}[I] = \emptyset \)
  - Repeat
    - For each instruction \( I \)
      - \( \text{in}[I] = (\text{out}[I] - \text{def}[I]) \cup \text{use}[I] \)
    - For each basic block \( B \)
      - \( \text{out}[B] = \bigcup_{B' \in \text{succ}(B)} \text{in}[B'] \)
  - Until no new changes in sets
Example

- Steps:
  - Set up live sets for each program point
  - Instantiate equations
  - Solve equations

Example

- Program points

Example

- Equation:
  - if (c)
  - x = y+1
  - y = 2*z
  - if (d)
  - x = y+z
  - z = 1
  - z = x

Example

- L1 = { x, y, z, c, d }
- L2 = { x, y, z, c, d }
- L3 = { x, y, z, c, d }
- L4 = { x, y, z, c, d }
- L5 = { x, y, z, c, d }
- L6 = { x, y, z, c, d }
- L7 = { x, y, z, c, d }
- L8 = { x, y, z, c, d }
- L9 = { x, y, z, c, d }
- L10 = { x, y, z, c, d }
- L11 = { x, y, z, c, d }
- L12 = {}

Questions

- Does this terminate?
- Does this compute the right answer?
- How could generalize this scheme for other kinds of analysis?

Generalization

- Dataflow analysis
  - A common framework for such analysis
  - Computes information at each program point
  - Conservative: characterizes all possible program behaviors

- Methodology
  - Describe the information (e.g., live variable sets) using a structure called a lattice
  - Build a system of equations based on:
    - How each statement affects information
    - How information flows between basic blocks
  - Solve the system of constraints

Parts of live variables analysis

- Live variable sets
  - Called flow values
  - Associated with program points
  - Start “empty”, eventually contain solution

- Effects of instructions
  - Called transfer functions
  - Take a flow value, compute a new flow value that captures the effects
  - One for each instruction – often a schema

- Handling control flow
  - Called confluence operator
  - Combines flow values from different paths
Mathematical model

- Flow values
  - Elements of a lattice $L = (P, \subseteq)$
  - Flow value $v \in P$
- Transfer functions
  - Set of functions (one for each instruction)
  - $F_i : P \rightarrow P$
- Confluence operator
  - Merges lattice values
  - $C : P \times P \rightarrow P$
  - How does this help us?

Lattices

- Lattice $L = (P, \subseteq)$
  - A partial order relation $\subseteq$
    - Reflexive, anti-symmetric, transitive
- Upper and lower bounds
  - Consider a subset $S$ of $P$
    - Upper bound of $S$: $u \in S \land \forall x \in S : x \subseteq u$
    - Lower bound of $S$: $l \in S \land \forall x \in S : l \subseteq x$
- Lattices are complete
  - Unique greatest and least elements
    - "Top": $T \in P \land \forall x \in P : x \subseteq T$
    - "Bottom": $\bot \in P \land \forall x \in P : \bot \subseteq x$

Confluence operator

- Combine flow values
  - "Merge" values on different control-flow paths
  - Result should be a safe over-approximation
  - We use the lattice $\subseteq$ to denote "more safe"

  Example: live variables
  - $v_1 = \{x, y, z\}$ and $v_2 = \{y, w\}$
  - How do we combine these values?
  - $v = v_1 \cup v_2 = \{w, x, y, z\}$
  - What is the "$\subseteq$" operator?
    - Superset

Meet and join

- Goal:
  - Combine two values to produce the "best" approximation
  - Intuition:
    - Given $v_1 = \{x, y, z\}$ and $v_2 = \{y, w\}$
    - A safe over-approximation is "all variables live"
    - We want the smallest set
  - Greatest lower bound
    - Given $x, y \in P$
    - GLB($x, y$) = $z$ such that $z \subseteq x$ and $z \subseteq y$ and
    - $\forall w : w \subseteq x$ and $w \subseteq y \Rightarrow w \subseteq z$
  - Meet operator: $x \land y = \text{GLB}(x, y)$
  - Natural "opposite": Least upper bound, $\text{join}$ operator

Termination

- Monotonicity
  - Transfer functions $F$ are monotonic if
    - Given $x, y \in P$
    - If $x \subseteq y$ then $F(x) \subseteq F(y)$
    - Alternatively: $F(x) \subseteq x$
  - Key idea:
    - Iterative dataflow analysis terminates if
      - Transfer functions are monotonic
      - Lattice has finite height
      - Intuition: values only go down, can only go to bottom

Example

- Prove monotonicity of live variables analysis
  - Equation: $\text{in}[i] = (\text{out}[i] - \text{def}[i]) \cup \text{use}[i]$
    (For each instruction $i$)
  - As a function: $F(x) = (x - \text{def}[i]) \cup \text{use}[i]$
  - Obligation: If $x \subseteq y$ then $F(x) \subseteq F(y)$
  - Proof:
    - $x \subseteq y$ $\Rightarrow$ $(x - \text{def}[i]) \cup \text{use}[i] \subseteq (y - \text{def}[i]) \cup \text{use}[i]$
    - Somewhat trivially:
      - $x \cup y \rightarrow x \cup s \subseteq y \cup s$
Dataflow solution

- Question:
  - What is the solution we compute?
  - Start at lattice top, move down
  - Called greatest \textit{fixpoint}
  - Where does approximation come from?
  - Confluence of control-flow paths
- Ideal solution?
  - Consider each path to a program point separately
  - Combine values at end
  - Called \textit{meet-over-all-paths} solution (MOP)
  - When is the fixpoint equal to MOP?

Composition of functions

Consider if-then-else graph
- If we compute each path:
  - $\text{in} = F_4(F_2(F_1(\text{out})))$
  - $\text{in} = F_4(F_3(F_1(\text{out})))$
- Two solutions
  - MOP:
    - $\text{in} = F_4(F_2(F_1(\text{out}))) \land F_4(F_3(F_1(\text{out})))$
  - Fixpoint:
    - Merge live vars before applying $F_4$
    - $\text{in} = F_4(F_2(F_1(\text{out}))) \land F_3(F_1(\text{out}))$
- When are these two results the same?
  - When the transfer functions are \textit{distributive}
  - Prove: $F(x) \land F(y) = F(x \land y)$

Summary

- Dataflow analysis
  - Lattice of flow values
  - Transfer functions (encode program behavior)
  - Iterative fixpoint computation
- Key insight:
  - If our dataflow equations have these properties:
    - Transfer functions are monotonic
    - Lattice has finite height
    - Transfer functions distribute over meet operator
  - Then:
    - Our fixpoint computation will terminate
    - Will compute \textit{meet-over-all-paths} solution

Examples

- Constant propagation
- Constant folding
- Common subexpression elimination
  (Available expressions)