EECS 483: Compiler Construction

Lecture 18:
Optimization and Dataflow Analysis

March 23
Winter Semester 2025

Slides adapted from Steve Zdancewic

Announcements

- Exam Grading almost done
- Assignment 4 due next Friday, April 4

Why optimize?

OPTIMIZATIONS, GENERALLY

When to apply optimization

High level
- AST
- IR

Mid level
- Canonical IR

Low level
- Abstract assembly
- Assembly

- Inlining
- Function specialization
- Constant folding
- Constant propagation
- Value numbering
- Dead code elimination
- Loop-invariant code motion
- Common sub-expression elimination
- Strength Reduction
- Constant folding & propagation
- Branch prediction / optimization
- Register allocation
- Loop unrolling
- Cache optimization

Safety

- Whether an optimization is *safe* depends on the programming language semantics.
  - Languages that provide weaker guarantees to the programmer permit more optimizations but have more ambiguity in their behavior.
  - *e.g.*, In C, loading from initialized memory is undefined, so the compiler can do anything if a program reads uninitalized data.
  - *e.g.*, In Java tail-call optimization (which turns recursive function calls into loops) is not valid because of “stack inspection”.
- Example: *loop-invariant code motion*
  - Idea: hoist invariant code out of a loop

```
while (b) {
  z = y/x;
  …                        // y, x not updated
}
```

```
z = y/x;
while (b) {
  …                        // y, x not updated
}
```

- Is this more efficient?
- Is this safe?

A high-level tour of a variety of optimizations.

BASIC OPTIMIZATIONS

Constant Folding

- Idea: If operands are known at compile type, perform the operation statically.

int x = (2 + 3) * y ➔ int x = 5 * y

b & false ➔ false

- Performed at every stage of optimization…
- Why?
  - Constant expressions can be created by translation or earlier optimizations

    Example: A[2] might be compiled to:

MEM[MEM[A] + 2 * 4] ➔ MEM[MEM[A] + 8]

Constant Folding Conditionals

```
if (true) S             → S
if (false) S            → ;
if (true) S else S'     → S
if (false) S else S'    → S'
while (false) S         → ;

if (2 > 3) S            →
    if (false) S            →   ;
```

Algebraic Simplification

- More general form of constant folding
  - Take advantage of mathematically sound simplification rules

- ***Mathematical identities***:
  - a * 1 ➔ a   a * 0 ➔ 0
  - a + 0 ➔ a   a – 0 ➔ a
  - b | false ➔ b   b & true ➔ b
- ***Reassociation & commutativity***:
  - (a + 1) + 2 ➔ a + (1 + 2) ➔ a + 3
  - (2 + a) + 4 ➔ (a + 2) + 4 ➔ a + (2 + 4) ➔ a + 6
- ***Strength reduction***: (replace expensive op with cheaper op)
  - a * 4 ➔ a << 2
  - a * 7 ➔ (a << 3) – a
  - a / 32767 ➔ (a >> 15) + (a >> 30)

- *Note 1:* must be careful with floating point (due to rounding) and integer arithmetic (due to overflow/underflow)
- *Note 2:* iteration of these optimizations is useful… how much?
- *Note 3:* must be sure that rewrites terminate:
  - commutativity apply like: (x + y) ➔ (y + x) ➔ (x + y) ➔ (y + x) ➔ …

Constant Propagation

- If a variable is known to be a constant, replace the use of the variable by the constant
- Value of the variable must be propagated forward from the point of assignment
  – This is a *substitution* operation

Example:

```
int x = 5;
int y = x * 2;
int z = a[y];
```

→

```
int y = 5 * 2;
int z = a[y];
```

→

```
int y = 10;
int z = a[y];
```

→

```
int z = a[10];
```

- To be most effective, constant propagation should be interleaved with constant folding

Copy Propagation

- If one variable is assigned to another, replace uses of the assigned variable with the copied variable.
- Need to know where copies of the variable propagate.
- Interacts with the scoping rules of the language.

- Example:

```
x = y;
if (x > 1) {
  x = x * f(x – 1);
}
```

➔

```
~~x = y;~~
if (y > 1) {
  x = y * f(y – 1);
}
```

- Can make the first assignment to x ***dead code*** (that can be eliminated).

Dead Code Elimination

- If a side-effect free statement can never be observed, it is safe to eliminate the statement.

```
x  = y * y  // x is dead!
…           // x never used
x = z * z
```

```
→             …
 x = z * z
```

- A variable is ***dead*** if it is never used after it is defined.
  – Computing such *definition* and *use* information is an important component of program analysis

- Dead variables can be created by other optimizations…

Unreachable/Dead Code

- Basic blocks not reachable by any trace leading from the starting basic block are *unreachable* and can be deleted.
  - Performed at the IR or assembly level
  - Improves cache, TLB performance

- Dead code: similar to unreachable blocks.
  - A value might be computed but never subsequently used.
- Code for computing the value can be dropped
- But only if it's ***pure***, *i.e.*, it has *no externally visible side effects*
  - Externally visible effects: raising an exception, modifying a global variable, going into an infinite loop, printing to standard output, sending a network packet, launching a rocket
  - Note: Pure functional languages (*e.g.*, Haskell) make reasoning about the safety of optimizations (and code transformations in general) easier!

Inlining

- Replace a call to a function with the body of the function itself with arguments rewritten to be local variables:
- Example in C: inline pow into g

```
int g(int x) { return x + pow(x); }
int pow(int a) {
        var b = 1; var x = 0;
        while (x < a) {b = 2 * b; x = x + 1}
        return b;
}
```

note: renaming

➔

```
int g(int x) {
        int a = x;
        int b = 1; int x2 = 0;
        while (x2 < a) {b = 2 * b; x2 = x2 + 1};
        tmp = b;
        return x + tmp;
}
```

- May need to rename variables to avoid *capture*
- Best done at the AST or relatively high-level IR.
- When is it profitable?
  - Eliminates the stack manipulation, jump, etc.
  - Can increase code size.
  - Enables further optimizations

Code Specialization

- Idea: create specialized versions of a function that is called from different places with different arguments.
- Example: specialize function f in:

```
class A implements I { int m() {…} }
class B implements I { int m() {…} }
int f(I x) { x.m(); }            // don't know which m
A a = new A(); f(a);             // know it's A.m
B b = new B(); f(b);             // know it's B.m
```

- f_A would have code specialized to dispatch to A.m
- f_B would have code specialized to dispatch to B.m
- You can also inline methods when the run-time type is known statically
  – Often just one class implements a method.

Common Subexpression Elimination

- *fold redundant computations together*
  - in some sense, it's the opposite of inlining
- Example:

```
a[i] = a[i] + 1
```

compiles to:

```
[a + i*4] = [a + i*4] + 1
```

Common subexpression elimination removes the redundant add and multiply:

```
t = a + i*4; [t] = [t] + 1
```

- For safety, you must be sure that the shared expression always has the same value in both places!

Unsafe Common Subexpression Elimination

- Example: consider this C function:

```
unit f(int[] a, int[] b, int[] c) {
    var j = …; var i = …; var k = …;
    b[j] = a[i] + 1;
    c[k] = a[i];
    return;
}
```

- The optimization that shares the expression `a[i]` is unsafe… why?

```
unit f(int[] a, int[] b, int[] c) {
    var j = …; var i = …; var k = …;
    t = a[i];
    b[j] = t + 1;
    c[k] = t;
    return;
}
```

LOOP OPTIMIZATIONS

Loop Optimizations

- Program hot spots often occur in loops.
  - Especially inner loops
  - Not always: consider operating systems code or compilers vs. a computer game or word processor
- Most program execution time occurs in loops.
  - The 90/10 rule of thumb holds here too.
    (90% of the execution time is spent in 10% of the code)
- Loop optimizations are very important, effective, and numerous
  - Also, concentrating effort to improve loop body code is usually a win

Loop Invariant Code Motion (revisited)

- Another form of redundancy elimination.
- If the result of a statement or expression does not change during the loop *and* it's pure, it can be hoisted outside the loop body.
- Often useful for array element addressing code
  - Invariant code not visible at the source level

```
for (i = 0; i < a.length; i++) {
  /* a not modified in the body */
}
```

```
t = a.length;
for (i =0; i < t; i++) {
  /* same body as above */
}
```

Hoisted loop-invariant expression

Strength Reduction (revisited)

- Strength reduction can work for loops too
- Idea: replace expensive operations (multiplies, divides) by cheap ones (adds and subtracts)
- For loops, create a *dependent induction variable*:

- Example:

```
for (int i = 0; i<n; i++) { a[i*3] = 1; }  // stride by 3
```

```
int j = 0;
for (int i = 0; i<n; i++) {
  a[j] = 1;
  j = j + 3;    // replace multiply by add
}
```

Loop Unrolling (revisited)

- Branches can be expensive, unroll loops to avoid them.

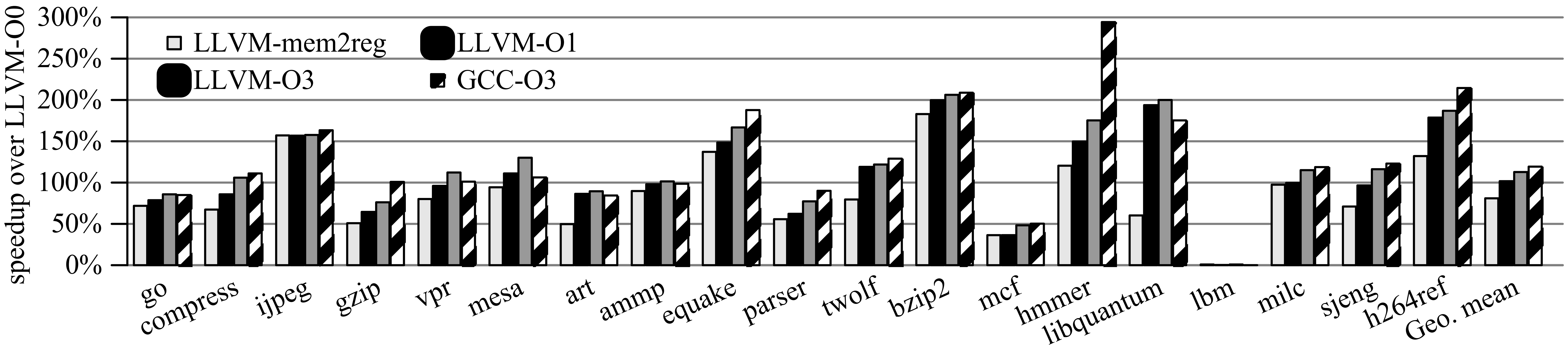
```
for (int i=0; i<n; i++) { S }
```

```
for (int i=0; i<n-3; i+=4) {S;S;S;S};
for (       ; i<n; i++) { S }  // left over iterations
```

- With k unrollings, eliminates (k-1)/k conditional branches
  - So for the above program, it eliminates ¾ of the branches
- Space-time tradeoff:
  - Not a good idea for large S or small n
- Interacts with instruction caching, branch prediction

EFFECTIVENESS?

Optimization Effectiveness?



$$\%\text{speedup} = \left[\frac{\text{base time}}{\text{optimized time}} - 1\right] \times 100\%$$

Example:

base time = 2s

optimized time = 1s $\Rightarrow$ 100% speedup

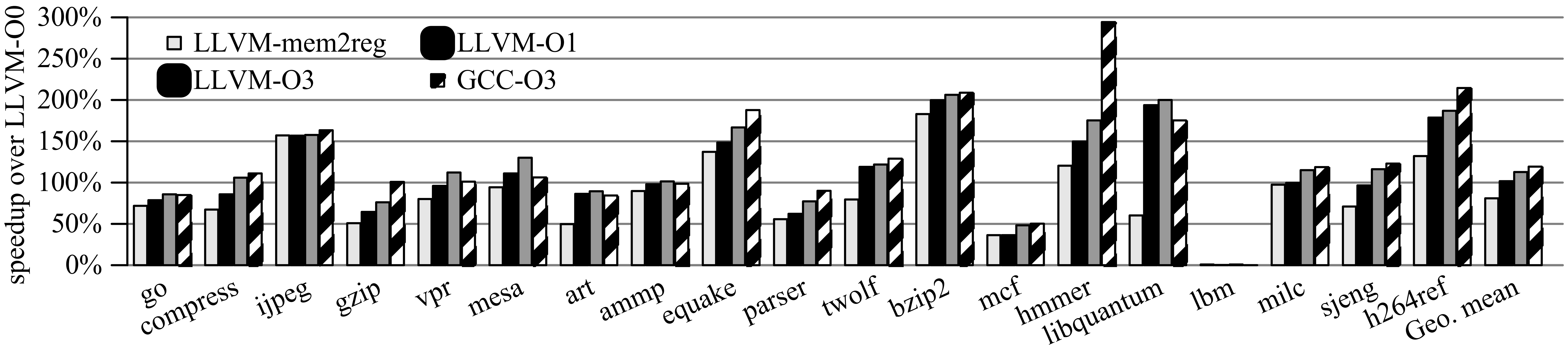
Example:

base time = 1.2s

optimized time = 0.87s $\Rightarrow$ 38% speedup

Graph taken from:
Jianzhou Zhao, Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic.
Formal Verification of SSA-Based Optimizations for LLVM.
In Proc. 2013 ACM SIGPLAN Conference on Programming Languages Design and Implementation (PLDI), 2013

Optimization Effectiveness?



- mem2reg: promotes alloca'ed stack slots to temporaries to enable register allocation
- Analysis:
  - mem2reg alone (+ back-end optimizations like register allocation) yields ~78% speedup on average
  - -O1 yields ~100% speedup
    (so all the rest of the optimizations combined account for ~22%)
  - -O3 yields ~120% speedup
- Hypothetical program that takes 10 sec. (base time):
  - Mem2reg alone: expect ~5.6 sec
  - -O1: expect ~5 sec
  - -O3: expect ~4.5 sec

CODE ANALYSIS

Motivating Code Analyses

- There are lots of things that might influence the safety/applicability of an optimization
  - What algorithms and data structures can help?

- How do you know what is a loop?
- How do you know an expression is invariant?
- How do you know if an expression has no side effects?
- How do you keep track of where a variable is defined?
- How do you know where a variable is used?
- How do you know if two reference values may be aliases of one another?

Control-flow Graphs Revisited

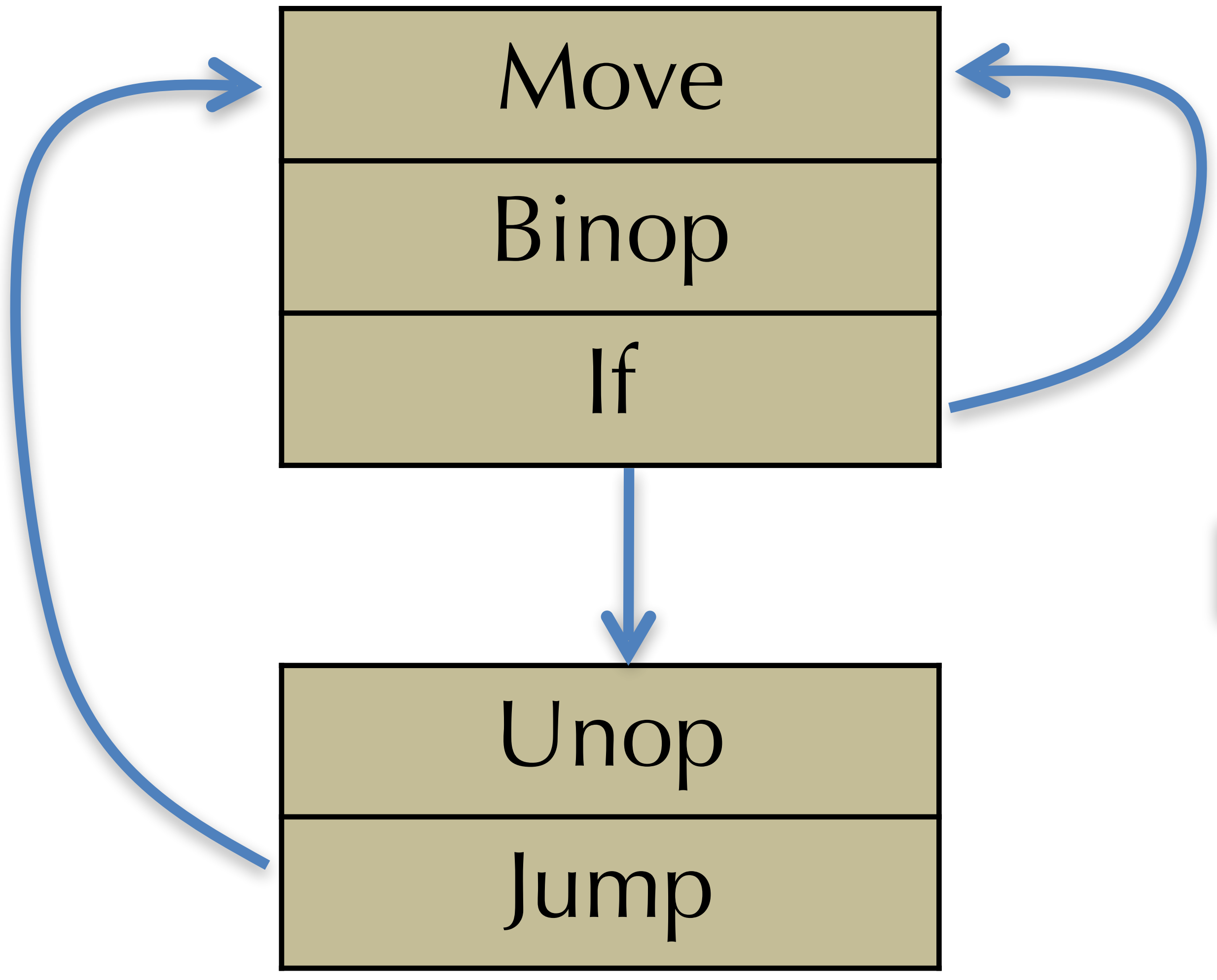
- For the purposes of dataflow analysis, we use the *control-flow graph* (CFG) intermediate form.
- Recall that a basic block is a sequence of instructions such that:
  - There is a distinguished, labeled entry point (no jumps into the middle of a basic block)
  - There is a (possibly empty) sequence of non-control-flow instructions
  - The block ends with a single control-flow instruction (jump, conditional branch, return, etc.)

- A *control flow graph*
  - Nodes are blocks
  - There is an edge from B1 to B2 if the control-flow instruction of B1 might jump to the entry label of B2
  - There are no "dangling" edges – there is a block for every jump target.

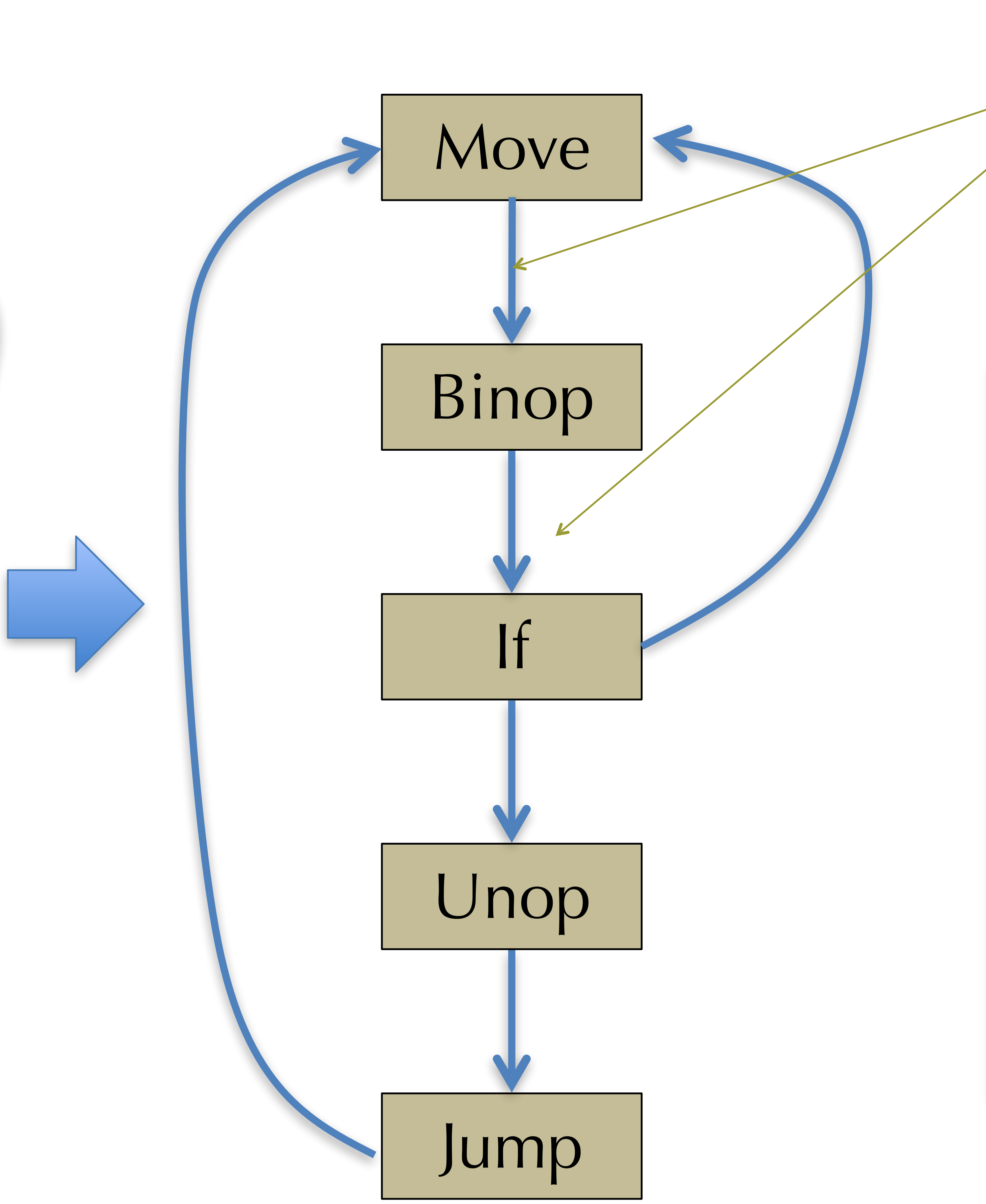
- Note: the following slides are intentionally a bit ambiguous about the exact nature of the code in the control flow graphs:
  - at the x86 assembly level
  - an "imperative" C-like source level
  - at the LLVM IR level
  - Same general idea, but the exact details will differ
    - e.g. LLVM IR doesn't have "imperative" update of %uid temporaries.
    - In fact, the SSA structure of the LLVM IR makes some of these analyses simpler.

Dataflow over CFGs

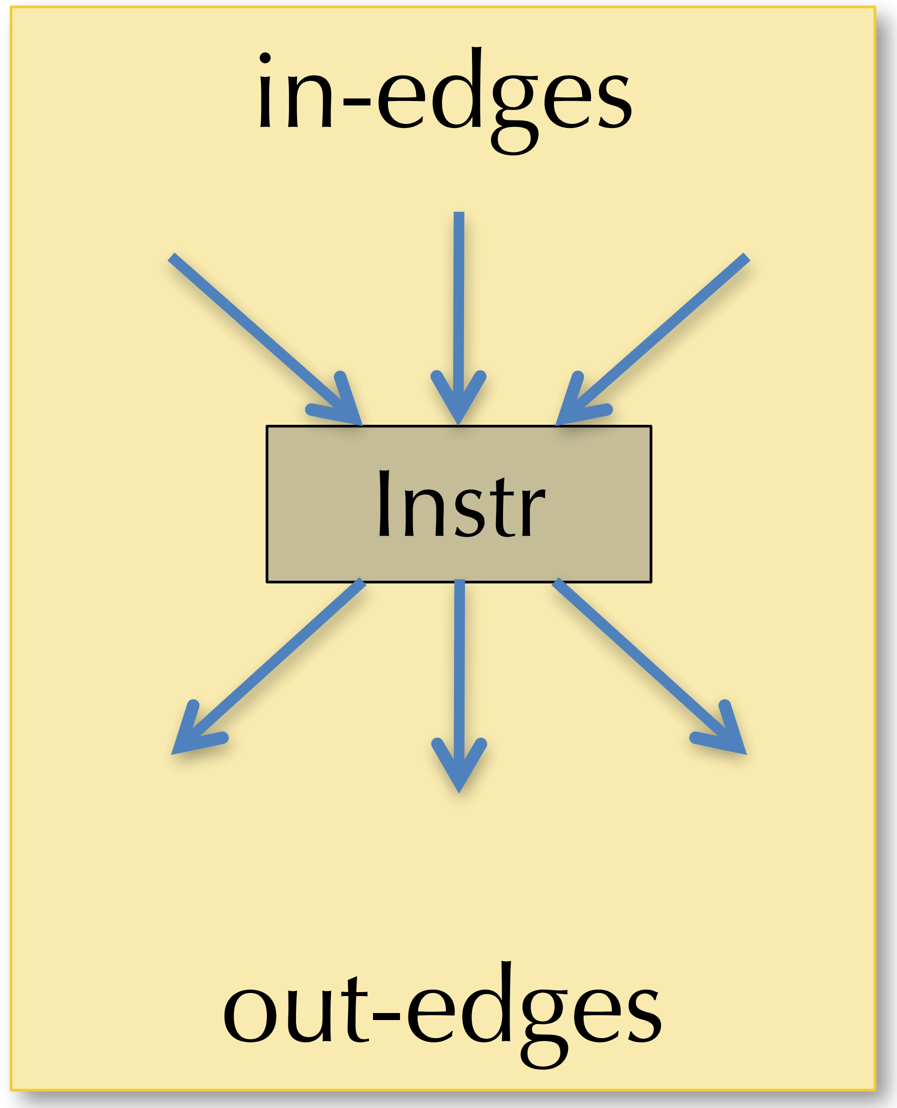
- For precision, it is helpful to think of the “fall through” between sequential instructions as an edge of the control-flow graph too.
  - Different implementation tradeoffs in practice…



Basic block CFG



“Exploded” CFG



Dataflow Analysis

- *Idea*: compute liveness information for all variables simultaneously.
  - Keep track of sets of information about each node

- Approach: define *equations* that must be satisfied by any liveness determination.
  - Equations based on "obvious" constraints.

- Solve the equations by iteratively converging on a solution.
  - Start with a "rough" approximation to the answer
  - Refine the answer at each iteration
  - Keep going until no more refinement is possible: a *fixpoint* has been reached

- This is an instance of a general framework for computing program properties: dataflow analysis

Dataflow Value Sets for Liveness

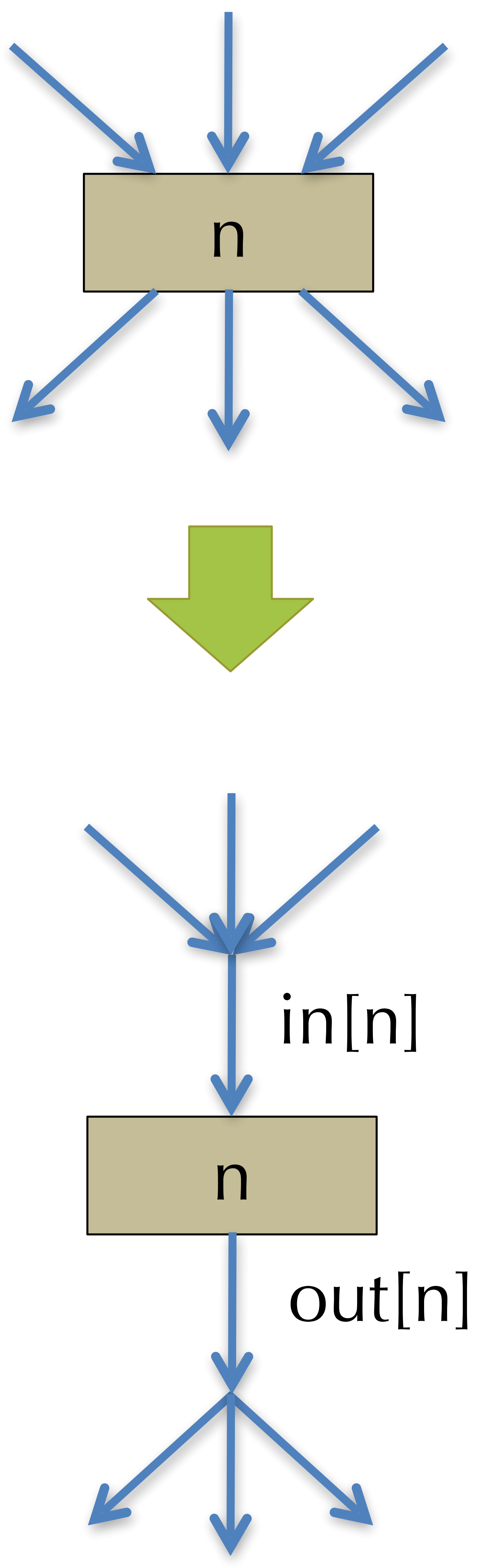
- Nodes are program statements, so:
- use[n] : set of variables used by n
- def[n] : set of variables defined by n
- in[n] : set of variables live on entry to n
- out[n] : set of variables live on exit from n

- Associate in[n] and out[n] with the "collected" information about incoming/outgoing edges

- For Liveness: what constraints are there among these sets?
- Clearly:

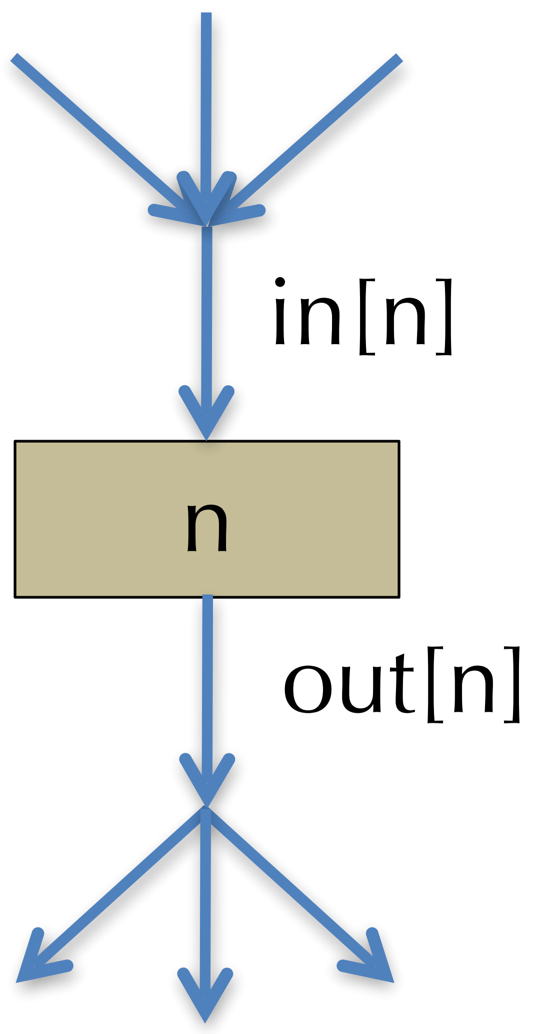
$$in[n] \supseteq use[n]$$

- What other constraints?



Other Dataflow Constraints

- We have: in[n] ⊇ use[n]
  - "A variable must be live on entry to n if it is used by n"
- Also: in[n] ⊇ out[n] - def[n]
  - "If a variable is live on exit from n, and n doesn't define it, it is live on entry to n"
  - Note: here '-' means "set difference"
- And: out[n] ⊇ in[n′] if n′ ∈ succ[n]
  - "If a variable is live on entry to a successor node of n, it must be live on exit from n."



Iterative Dataflow Analysis

- Find a solution to those constraints by starting from a rough guess.
  - Start with: $in[n] = \emptyset$ and $out[n] = \emptyset$
- The guesses don't satisfy the constraints:
  - $in[n] \supseteq use[n]$
  - $in[n] \supseteq out[n] - def[n]$
  - $out[n] \supseteq in[n']$ if $n' \in succ[n]$

- Idea: iteratively re-compute in[n] and out[n] where forced to by the constraints.
  - Each iteration will add variables to the sets in[n] and out[n] (i.e. the live variable sets will increase monotonically)
- We stop when in[n] and out[n] satisfy these equations: (which are derived from the constraints above)
  - $in[n] = use[n] \cup (out[n] - def[n])$
  - $out[n] = \bigcup_{n' \in succ[n]} in[n']$
    - Most SSA constructs: 1 successor
      - ret has 0 successors: its out set is empty
      - cbr has 2 successors: its out set is the **union** of the two out sets

Complete Liveness Analysis Algorithm

for all n, in[n] := Ø, out[n] := Ø

repeat until no change in 'in' and 'out'

    for all n

        $out[n] := \bigcup_{n' \in succ[n]} in[n']$

        $in[n] := use[n] \cup (out[n] - def[n])$

    end

end

- Finds a *fixpoint* of the in and out equations.
  - The algorithm is guaranteed to terminate… Why?
- Why do we start with Ø?

“Classic” Constant Propagation

- Constant propagation can be formulated as a dataflow analysis.

- Idea: propagate and fold integer constants in one pass:

```
x = 1;              x = 1;
y = 5 + x;          y = 6;
z = y * y;          z = 36;
```

- Information about a single variable:
  - Variable is never defined.
  - Variable has a single, constant value.
  - Variable is assigned multiple values.

Assertion Removal

- Dynamic typing adds many runtime assertions into our program.

- ```
  let x = g() in
  let y = x + 2 in
  let z = y * x in
  ...
  ```
- Current compilation always adds assertions that inputs are integers

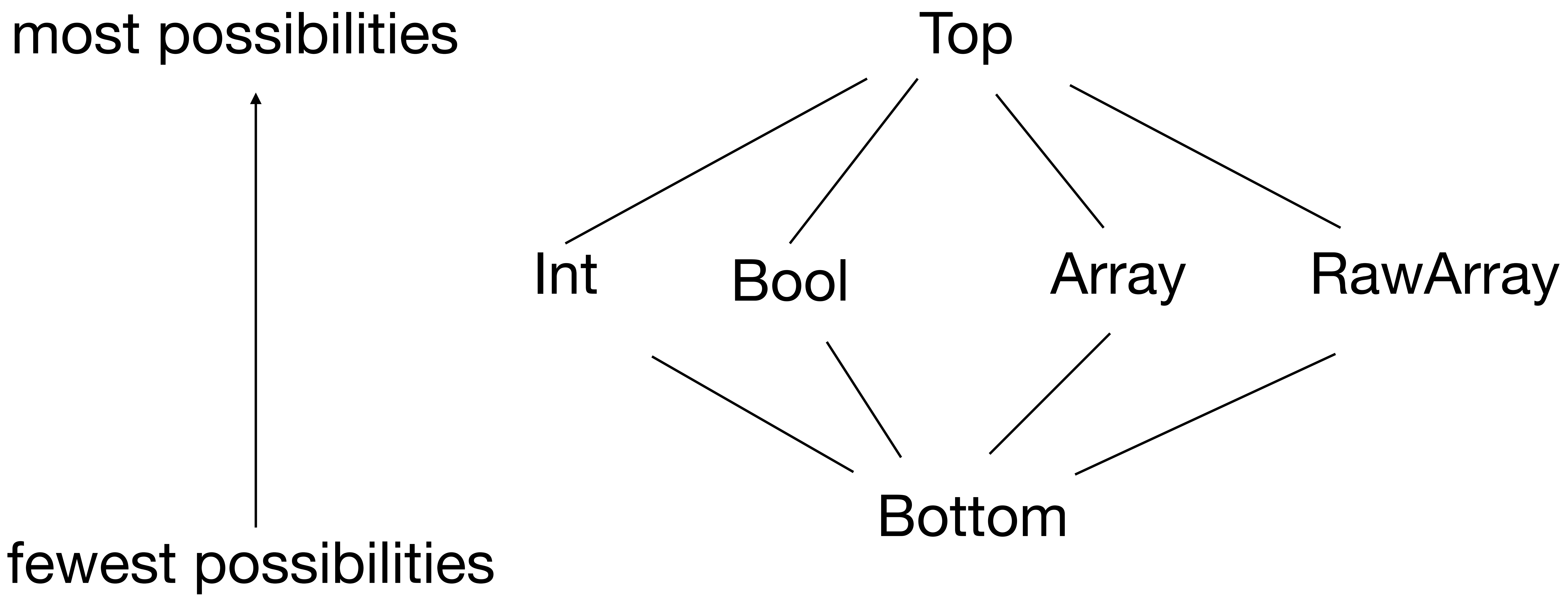
- ```
  x = f()
  assertInt(x)
  y = x + 2
  assertInt(y)
  assertInt(x)
  y2 = y >> 1
  z = y2 * x
  ...
  ```
- Which assertions can we remove?

Tag-checking Analysis

- At each program point, for each variable associate an approximation of what the possible values are:
  - Int: tagged integer, i.e., multiple of 2
  - Bool: tagged boolean, i.e., either 0b001 or 0b101
  - RawArray: untagged pointer to an array on the heap
  - Array: tagged array, i.e., a pointer tagged with 0b11
  - Top: any 64 bit value
  - Bottom: never assigned to, i.e., uninitialized
- Usage: If analysis determines x is an Int, then remove assertions assertInt(x)

  similar for assertArray, assertBool etc.

Tag-checking Analysis



Straightline Code Example

```
x = f()
assertInt(x)
y = x + 2
assertInt(y)
assertInt(x)
y2 = y >> 1
z = y2 * x
```

Tag-checking Analysis

- For each operation in SSA, need to define "flow function" that says what possible tags are based on inputs.

  Examples:
- x = y + z
  - if y and z are tagged Ints, then x is a tagged Int
  - otherwise x is Top
- x = y * z
  - if y or z is a tagged Int then x is a tagged Int
  - otherwise Top
- x = y << n
  - if n is at least 1 then x is tagged
  - if n is 0, then x is tagged if y is
- assertInt(x)
  - after this, x is always a tagged Int, because otherwise execution ended

Straightline Code Example

```
0
x = f()
1assertInt(x)
2y = x + 2
3assertInt(y)
4assertInt(x)
5y2 = y >> 1
6
z = y2 * x
7
```

```
0:
1: {x: Top}
2: {x: Int}
3: {x: Int, y: Int}
4: {x: Int, y: Int}
5: {x: Int, y: Int}
6: {x: Int, y: Int, y2: Top}
7: {x: Int, y: Int, y2: Top, z: Int}
```

Straightline Code Example

0
`x = f()`
1 `assertInt(x)`
2 `y = x + 2`
3 ~~`assertInt(y)`~~
4 ~~`assertInt(x)`~~
5 `y2 = y >> 1`
6 `z = y2 * x`
7

```
0:
1: {x: Top}
2: {x: Int}
3: {x: Int, y: Int}
4: {x: Int, y: Int}
5: {x: Int, y: Int}
6: {x: Int, y: Int, y2: Top}
7: {x: Int, y: Int, y2: Top, z: Int}
```

Loop Example

```
extern g
def f(y,z):
  def loop(i,a):
    if i == 0:
      a - z
    else:
      loop(i - 1, a + g())
  in
  loop(y, 0)
```

```
f(y,z):
  loop(i,a):
    thn():
      assertInt(a)
      assertInt(z)
      r = a - z
      ret r
    els():
      assertInt(i)
      i' = i - 2
      x = g()
      assertInt(a)
      assertInt(x)
      a' = a + x
      br loop(i', a')
    b = i == 0
    cbr b thn() els()
  br loop(y, 0)
```

Loop Example

```
f(y,z):
  loop(i,a):
    thn():
      assertInt(a)
      assertInt(z)
      r = a - z
      ret r
    els():
      assertInt(i)
      i' = i - 2
      x = g()
      assertInt(a)
      assertInt(x)
      a' = a + x
      br loop(i', a')
    b = i == 0
    cbr b thn() els()
  br loop(y, 0)
```

What information do we use in internal blocks such as **loop, thn, els**?

- These are **recursive**, same problem as liveness
- Same solution: initialize their info to bottom, iteratively update them
- Main difference: need to compute info based on **predecessors** of the block

What about top-level functions like **f?**

- Initialize its inputs to **top** as they can be called with any snake values
- In Diamondback, there is only one top-level function, **main(x),** and its input is an **Array**.

Tag-checking Analysis

- When analysing a block f(x1,...): what information do we know about the parameters?
  - The value of x1 could be the value of any a1 for any br f(a1,...) in the program. These are the "predecessors" of f.
  - information for a parameter x1 should be the best approximation of the information known for all instantiations a1
  - roughly the "union" of all possible values it could be, i.e., the "least upper bound"
- Examples:
  - f(x): ...
  - br f(a) and br f(b) where a: Int and b: Int then x: Int
  - br f(a) and br f(b) where a: Int and b: Bool then x: Top
  - generally: if all predeccesors agree on the tag, then use that tag, if any disagree, use Top

```
f(y,z):
  loop(i,a):
    thn():
     3 assertInt(a)
     4 assertInt(z)
     5 r = a - z
     6 ret r
    els():
     7 assertInt(i)
     8 i' = i - 2
     9 x = g()
    10 assertInt(a)
    11 assertInt(x)
    12 a' = a + x
    13 br loop(i', a')
   1 b = i == 0
   2 cbr b thn() els()
  0 br loop(y, 0)
```

```
f(y,z):
  loop(i,a):
    thn():
     3 assertInt(a)
     4 assertInt(z)
     5 r = a - z
     6 ret r
    els():
     7 assertInt(i)
     8 i' = i - 2
     9 x = g()
    10 assertInt(a)
    11 assertInt(x)
    12 a' = a + x
    13 br loop(i', a')
   1 b = i == 0
   2 cbr b thn() els()
  0 br loop(y, 0)
```

unmentioned vars: bottom

```
 0: {y:Top,z:Top}
 1: bottom
 2:
 3: bottom
 4:
 5:
 6:
 7: bottom
 8:
 9:
10:
11:
12:
13:
```

```
f(y,z):
  loop(i,a):
    thn():
     3assertInt(a)
     4assertInt(z)
     5r = a - z
     6ret r
    els():
     7assertInt(i)
     8i' = i - 2
     9x = g()
    10assertInt(a)
    11assertInt(x)
    12a' = a + x
    13br loop(i', a')
   1b = i == 0
   2cbr b thn() els()
  0br loop(y, 0)
```

unmentioned vars: bottom

```
0: {y:Top,z:Top}
1: bottom
2: {b:Top}
3: bottom
4: {a:Int}
5: {a:Int,z:Int}
6: {a:Int,z:Int,r:Int}
7: bottom
8: {i:Int}
9: {i:Int,i':Int}
10: {i:Int,i':Int,x:Top}
11: {i:Int,i':Int,x:Top,a:Int}
12: {i:Int,i':Int,x:Top,a:Int,x:Int}
13: {i:Int,i':Int,x:Top,a:Int,x:Int,a':Int}
```

```
f(y,z):
  loop(i,a):
    thn():
     3 assertInt(a)
     4 assertInt(z)
     5 r = a - z
     6 ret r
    els():
     7 assertInt(i)
     8 i' = i - 2
     9 x = g()
    10 assertInt(a)
    11 assertInt(x)
    12 a' = a + x
    13 br loop(i', a')
   1 b = i == 0
   2 cbr b thn() els()
 0 br loop(y, 0)
```

unmentioned vars: bottom

```
0:  {y:Top,z:Top}
1:  bottom
2:  {b:Top}
3:  bottom
4:  {a:Int}
5:  {a:Int,z:Int}
6:  {a:Int,z:Int,r:Int}
7:  bottom
8:  {i:Int}
9:  {i:Int,i':Int}
10: {i:Int,i':Int,x:Top}
11: {i:Int,i':Int,x:Top,a:Int}
12: {i:Int,i':Int,x:Top,a:Int,x:Int}
13: {i:Int,i':Int,x:Top,a:Int,x:Int,a':Int}
```

How do we update blocks (1,3,7) based on the previous round?

```
f(y,z):
  loop(i,a):
    thn():
     3assertInt(a)
     4assertInt(z)
     5r = a - z
     6ret r
    els():
     7assertInt(i)
     8i' = i - 2
     9x = g()
    10assertInt(a)
    11assertInt(x)
    12a' = a + x
    13br loop(i', a')
   1b = i == 0
   2cbr b thn() els()
  0br loop(y, 0)
```

Previous iteration output

```
0: {y:Top,z:Top}
1: bottom
13: {i:Int,i':Int,x:Top,a:Int,x:Int,a':Int}
```

Next iteration, update loop based on its two predecessors

```
1: {y:Top,z:Top,i = y:Top,a = 0:Int}
   U {i = i':Int,a = a':Int}
```

simplification: remove variables that aren't in scope

```
f(y,z):
  loop(i,a):
    thn():
     3 assertInt(a)
     4 assertInt(z)
     5 r = a - z
     6 ret r
    els():
     7 assertInt(i)
     8 i' = i - 2
     9 x = g()
    10 assertInt(a)
    11 assertInt(x)
    12 a' = a + x
    13 br loop(i', a')
   1 b = i == 0
   2 cbr b thn() els()
 0 br loop(y, 0)
```

Previous iteration output

```
 0: {y:Top,z:Top}
 1: bottom
13: {i:Int,i':Int,x:Top,a:Int,x:Int,a':Int}
```

Next iteration, update loop based on its two predecessors

```
1: {y:Top,z:Top,i:Top,a:Int}
```

```
f(y,z):
  loop(i,a):
    thn():
3     assertInt(a)
4     assertInt(z)
5     r = a - z
6     ret r
    els():
7     assertInt(i)
8     i' = i - 2
9     x = g()
10    assertInt(a)
11    assertInt(x)
12    a' = a + x
13    br loop(i', a')
1   b = i == 0
2   cbr b thn() els()
0 br loop(y, 0)
```

Converges after one round

```
0: {y:Top,z:Top}
1: {y:Top,z:Top,i:Top,a:Int}
2: {y:Top,z:Top,i:Top,a:Int,b:Top}
3: {y:Top,z:Top,i:Top,a:Int,b:Top}
4: {y:Top,z:Top,i:Top,a:Int,b:Top}
5: {y:Top,z:Int,i:Top,a:Int,b:Top}
6: {y:Top,z:Top,i:Top,a:Int,b:Top,r:Int}
7: {y:Top,z:Top,i:Top,a:Int,b:Top}
8: {y:Top,z:Top,i:Int,a:Int,b:Top}
9: {y:Top,z:Top,i:Int,a:Int,b:Top,i':Int}
10: {y:Top,z:Top,i:Int,a:Int,b:Top,i':Int,x:Top}
11: {y:Top,z:Top,i:Int,a:Int,b:Top,i':Int,x:Top}
12: {y:Top,z:Top,i:Int,a:Int,b:Top,i':Int,x:Int}
13: {y:Top,z:Top,i:Int,a:Int,b:Top,i':Int,x:Int,a':Int}
```

```
f(y,z):
  loop(i,a):
    thn():
     3 assertInt(a)
     4 assertInt(z)
     5 r = a - z
     6 ret r
    els():
     7 assertInt(i)
     8 i' = i - 2
     9 x = g()
    10 assertInt(a)
    11 assertInt(x)
    12 a' = a + x
    13 br loop(i', a')
   1 b = i == 0
   2 cbr b thn() els()
  0 br loop(y, 0)
```

remove assertions based on inferred info

```
0: {y:Top,z:Top}
1: {y:Top,z:Top,i:Top,a:Int}
2: {y:Top,z:Top,i:Top,a:Int,b:Top}
3: {y:Top,z:Top,i:Top,a:Int,b:Top}
4: {y:Top,z:Top,i:Top,a:Int,b:Top}
5: {y:Top,z:Int,i:Top,a:Int,b:Top}
6: {y:Top,z:Top,i:Top,a:Int,b:Top,r:Int}
7: {y:Top,z:Top,i:Top,a:Int,b:Top}
8: {y:Top,z:Top,i:Int,a:Int,b:Top}
9: {y:Top,z:Top,i:Int,a:Int,b:Top,i':Int}
10: {y:Top,z:Top,i:Int,a:Int,b:Top,i':Int,x:Top}
11: {y:Top,z:Top,i:Int,a:Int,b:Top,i':Int,x:Top}
12: {y:Top,z:Top,i:Int,a:Int,b:Top,i':Int,x:Int}
13: {y:Top,z:Top,i:Int,a:Int,b:Top,i':Int,x:Int,a':Int}
```

unfortunately, can't remove the assertInt(i) because we don't know initial value of your is an Int

```
extern g
def f(y,z):
  def loop(i,a):
    if i == 0:
      a - z
    else:
      loop(i - 1, a + g())
  in
  loop(y, 0)
```

inline once

```
extern g
def f(y,z):
  def loop(i,a):
    if i == 0:
      a - z
    else:
      loop(i - 1, a + g())
  in
  if y == 0:
   0 - z
  else:
    loop(y - 1, 0 + g())
```

with this change, now **i** will be determined to always be an Int

GENERAL DATAFLOW ANALYSIS

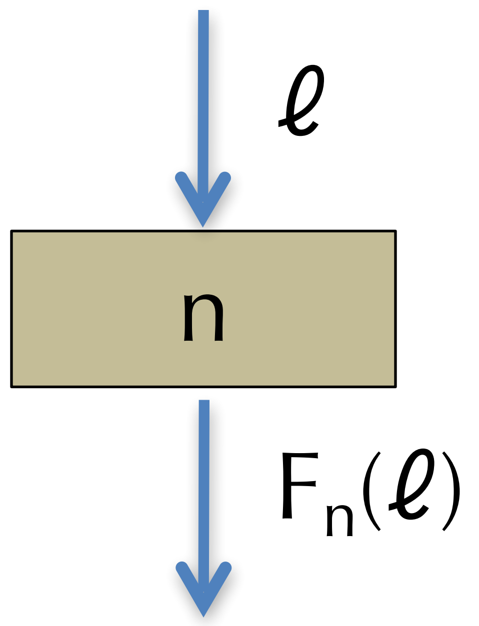
Common Features

- Analyses have a *domain* over which they solve constraints.
  - Liveness, the domain is sets of variables
  - Tag checking the domain is our tag sets
- Each analysis has a notion of flow function
  - Used to explain how information propagates across a node.
- Each analysis is propagates information either *forward* or *backward*
  - Forward: defined in terms of predecessor nodes
  - Backward: defined in terms of successor nodes
- Each analysis has a way of aggregating information
  - Liveness uses union ($\cup$)
  - Tag analysis uses least upper bound

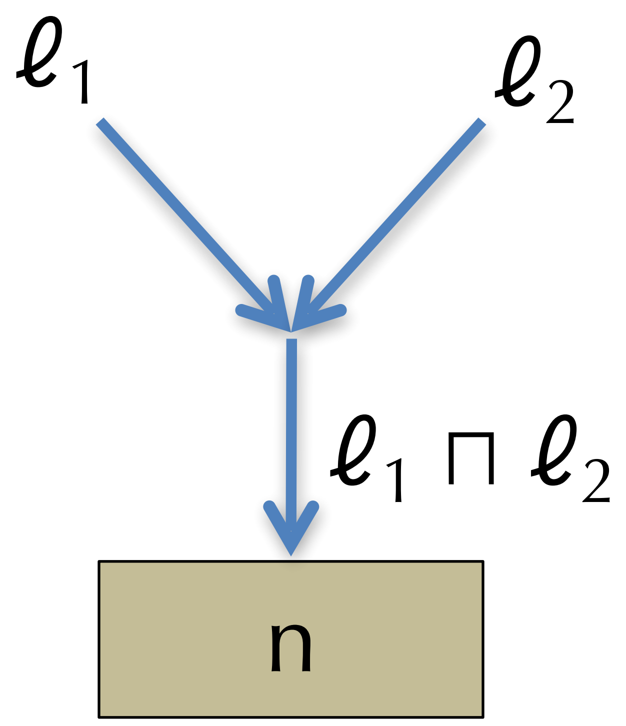
(Forward) Dataflow Analysis Framework

A forward dataflow analysis can be characterized by:

1. A domain of dataflow values L
   - e.g. L = the powerset of all variables
   - Think of $\ell \in L$ as a property, then "$x \in \ell$" means "x has the property"



2. For each node n, a flow function $F_n : L \rightarrow L$
   - "If ℓ is a property that holds before the node n, then $F_n(\ell)$ holds after n"
3. A combining operator $\sqcap$
   - "If we know *either* ℓ_1 *or* ℓ_2 holds on entry to node n, we know at most $\ell_1 \sqcap \ell_2$"
   - $in[n] := \bigsqcap_{n' \in pred[n]} out[n']$



Generic Iterative (Forward) Analysis

for all n, in[n] := $\top$, out[n] := $\top$

repeat until no change

    for all n

        $in[n] := \sqcap_{n' \in pred[n]} out[n']$

        $out[n] := F_n(in[n])$

    end

end

- Here, $\top \in L$ ("top") represents having the "maximum" amount of information.
  - Having "more" information enables more optimizations
  - "Maximum" amount could be inconsistent with the constraints.
  - Iteration refines the answer, eliminating inconsistencies

Structure of L

- The domain has structure that reflects the "amount" of information contained in each dataflow value.
- Some dataflow values are more informative than others:
  - Write $\ell_1 \sqsubseteq \ell_2$ whenever ℓ_2 provides at least as much information as ℓ_1.
  - The dataflow value ℓ_2 is "better" for enabling optimizations.

- Example 1: for liveness analysis, *smaller* sets of variables are more informative.
  - Having smaller sets of variables live across an edge means that there are fewer conflicts for register allocation assignments.
  - So: $\ell_1 \sqsubseteq \ell_2$ if and only if $\ell_1 \supseteq \ell_2$

- Example 2: for available expressions analysis, larger sets of nodes are more informative.
  - Having a larger set of nodes (equivalently, expressions) available means that there is more opportunity for common subexpression elimination.
  - So: $\ell_1 \sqsubseteq \ell_2$ if and only if $\ell_1 \subseteq \ell_2$

$\mathcal{L}$ as a Partial Order

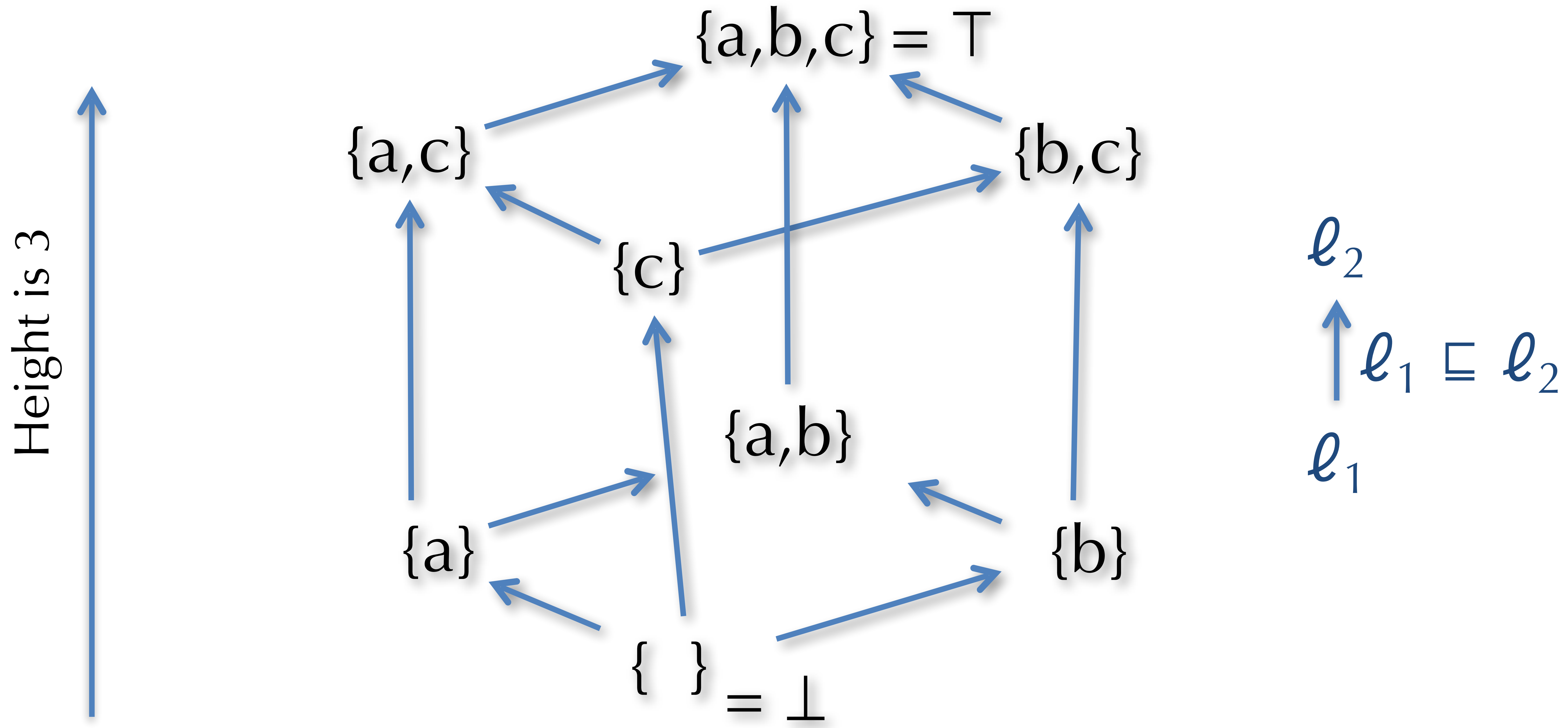
- $\mathcal{L}$ is a *partial order* defined by the ordering relation $\sqsubseteq$.
- A partial order is an ordered set.
- Some of the elements might be *incomparable*.
  - That is, there might be $\ell_1, \ell_2 \in \mathcal{L}$ such that neither $\ell_1 \sqsubseteq \ell_2$ nor $\ell_2 \sqsubseteq \ell_1$

- Properties of a partial order:
  - *Reflexivity:* $\ell \sqsubseteq \ell$
  - *Transitivity:* $\ell_1 \sqsubseteq \ell_2$ and $\ell_2 \sqsubseteq \ell_3$ implies $\ell_1 \sqsubseteq \ell_2$
  - *Anti-symmetry:* $\ell_1 \sqsubseteq \ell_2$ and $\ell_2 \sqsubseteq \ell_1$ implies $\ell_1 = \ell_2$

- Examples:
  - Integers ordered by $\leq$
  - Sets ordered by $\subseteq$ or $\supseteq$

Subsets of {a,b,c} ordered by ⊆

Partial order presented as a Hasse diagram.



order $\sqsubseteq$ is $\subseteq$ meet $\sqcap$ is $\cap$ join $\sqcup$ is $\cup$

Meets and Joins

- The combining operator $\sqcap$ is called the "meet" operation.
- It constructs the *greatest lower bound*:
  - $\ell_1 \sqcap \ell_2 \sqsubseteq \ell_1$ and $\ell_1 \sqcap \ell_2 \sqsubseteq \ell_2$
    "the meet is a lower bound"
  - If $\ell \sqsubseteq \ell_1$ and $\ell \sqsubseteq \ell_2$ then $\ell \sqsubseteq \ell_1 \sqcap \ell_2$
    "there is no greater lower bound"

- Dually, the $\sqcup$ operator is called the "join" operation.
- It constructs the *least upper bound*:
  - $\ell_1 \sqsubseteq \ell_1 \sqcup \ell_2$ and $\ell_2 \sqsubseteq \ell_1 \sqcup \ell_2$
    "the join is an upper bound"
  - If $\ell_1 \sqsubseteq \ell$ and $\ell_2 \sqsubseteq \ell$ then $\ell_1 \sqcup \ell_2 \sqsubseteq \ell$
    "there is no smaller upper bound"

- A partial order that has all meets and joins is called a *lattice*.
  - If it has just meets, it's called a *meet semi-lattice*.

Another Way to Describe the Algorithm

- Algorithm repeatedly computes (for each node n):
- out[n] := F_n(in[n])
- Equivalently: out[n] := $F_n(\sqcap_{n' \in pred[n]} out[n'])$
  - By definition of in[n]
- We can write this as a simultaneous update of the vector of out[n] values:
  - let x_n = out[n]
  - Let **X** = $(x_1, x_2, \ldots, x_n)$ it's a vector of points in $\mathcal{L}$
  - **F**(**X**) = $(F_1(\sqcap_{j \in pred[1]} out[j]), F_2(\sqcap_{j \in pred[2]} out[j]), \ldots, F_n(\sqcap_{j \in pred[n]} out[j]))$
- Any solution to the constraints is a *fixpoint* **X** of **F**
  - i.e. **F**(**X**) = **X**

Iteration Computes Fixpoints

- Let $\mathbf{X}_0 = (\top, \top, \ldots, \top)$
- Each loop through the algorithm apply F to the old vector:
  $\mathbf{X}_1 = \mathbf{F}(\mathbf{X}_0)$
  $\mathbf{X}_2 = \mathbf{F}(\mathbf{X}_1)$
  …
- $\mathbf{F}^{k+1}(\mathbf{X}) = \mathbf{F}(\mathbf{F}^{k}(\mathbf{X}))$
- A fixpoint is reached when $\mathbf{F}^{k}(\mathbf{X}) = \mathbf{F}^{k+1}(\mathbf{X})$
  - That's when the algorithm stops.
- Wanted: a maximal fixpoint
  - Because that one is more informative/useful for performing optimizations

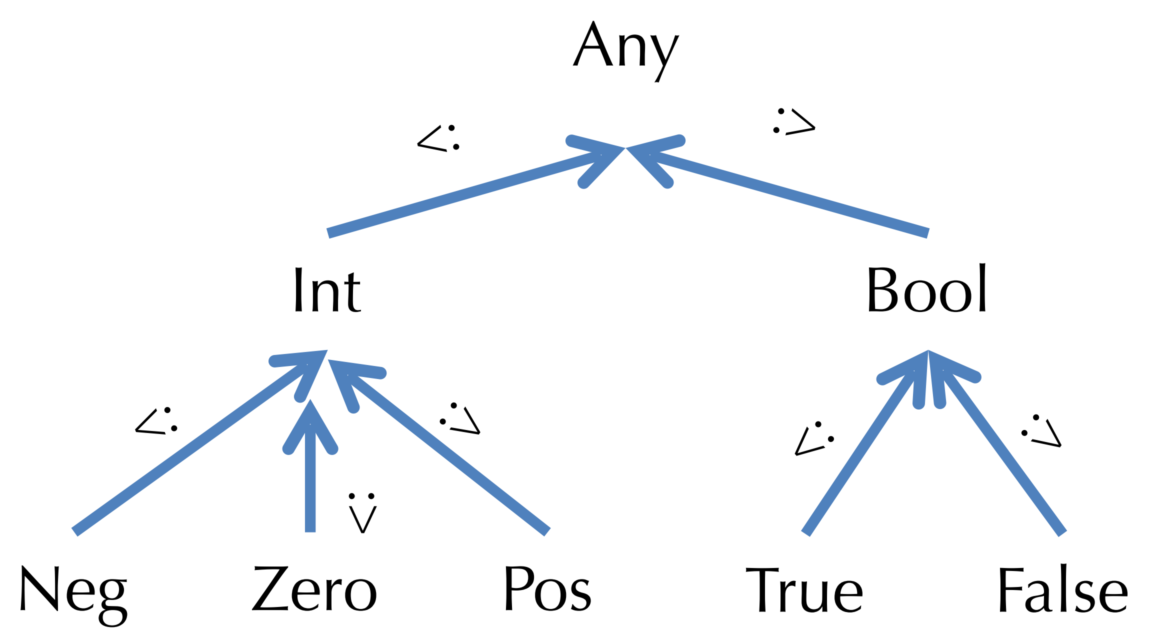
Monotonicity & Termination

- Each flow function F_n maps lattice elements to lattice elements; to be sensible is should be *monotonic*:
- $F : L \to L$ is *monotonic* iff:
  $\ell_1 \sqsubseteq \ell_2$ implies that $F(\ell_1) \sqsubseteq F(\ell_2)$
  - Intuitively: "If you have more information entering a node, then you have more information leaving the node."
- Monotonicity lifts point-wise to the function: $\mathbf{F} : L^n \to L^n$
  - vector $(x_1, x_2, \ldots, x_n) \sqsubseteq (y_1, y_2, \ldots, y_n)$ iff $x_i \sqsubseteq y_i$ for each i
- Note that **F** is consistent: $\mathbf{F}(\mathbf{X}_0) \sqsubseteq \mathbf{X}_0$
  - So each iteration moves at least one step down the lattice (for some component of the vector)
  - $\ldots \sqsubseteq \mathbf{F}(\mathbf{F}(\mathbf{X}_0)) \sqsubseteq \mathbf{F}(\mathbf{X}_0) \sqsubseteq \mathbf{X}_0$
- Therefore, # steps needed to reach a fixpoint is at most the height H of L times the number of nodes: O(Hn)

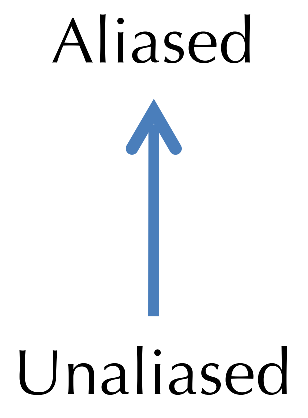
Building Lattices?

- Information about individual nodes or variables can be lifted *pointwise:*
  - If L is a lattice, then so is $\{ f : X \to L \}$ where $f \sqsubseteq g$ if and only if $f(x) \sqsubseteq g(x)$ for all $x \in X$.

- Like *types*, the dataflow lattices are *static approximations* to the dynamic behavior:
  - Could pick a lattice based on subtyping:



  - Or other information:



- Points in the lattice are sometimes called dataflow “facts”

"Classic" Constant Propagation

- Constant propagation can be formulated as a dataflow analysis.

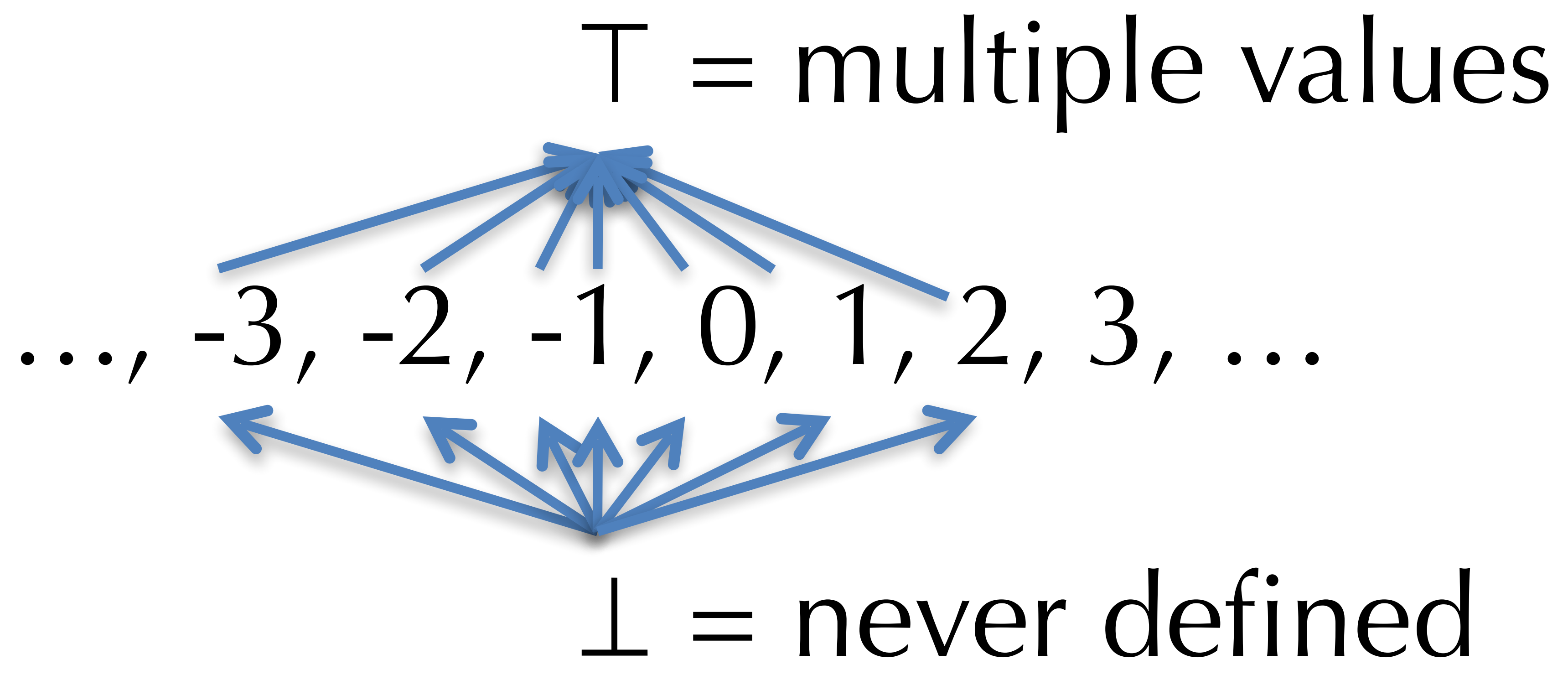
- Idea: propagate and fold integer constants in one pass:

x = 1;		x = 1;
y = 5 + x;	→	y = 6;
z = y * y;		z = 36;

- Information about a single variable:
  - Variable is never defined.
  - Variable has a single, constant value.
  - Variable is assigned multiple values.

Domains for Constant Propagation

- We can make a constant propagation lattice L for *one variable* like this:



- To accommodate multiple variables, we take the product lattice, with one element per variable.
  - Assuming there are three variables, x, y, and z, the elements of the product lattice are of the form (ℓ_x, ℓ_y, ℓ_z).
  - Alternatively, think of the product domain as a context that maps variable names to their "*abstract interpretations*"
- What are "meet" and "join" in this product lattice?
- What is the height of the product lattice?

Flow Functions

- Consider the node $x = y \text{ op } z$
- $F(\ell_x, \ell_y, \ell_z) = ?$

- $F(\ell_x, \top, \ell_z) = (\top, \top, \ell_z)$
- $F(\ell_x, \ell_y, \top) = (\top, \ell_y, \top)$

"If either input might have multiple values the result of the operation might too."

- $F(\ell_x, \perp, \ell_z) = (\perp, \perp, \ell_z)$
- $F(\ell_x, \ell_y, \perp) = (\perp, \ell_y, \perp)$

"If either input is undefined the result of the operation is too."

"If the inputs are known constants, calculate the output statically."

- $F(\ell_x, i, j) = (i \text{ op } j, i, j)$

- Flow functions for the other nodes are easy…
- Monotonic?

Dataflow Analysis: Summary

- Many dataflow analyses fit into a common framework.
- Key idea: *Iterative solution* of a system of equations over a *lattice* of constraints.
  - Iteration terminates if flow functions are monotonic.