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 uninitialized 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.

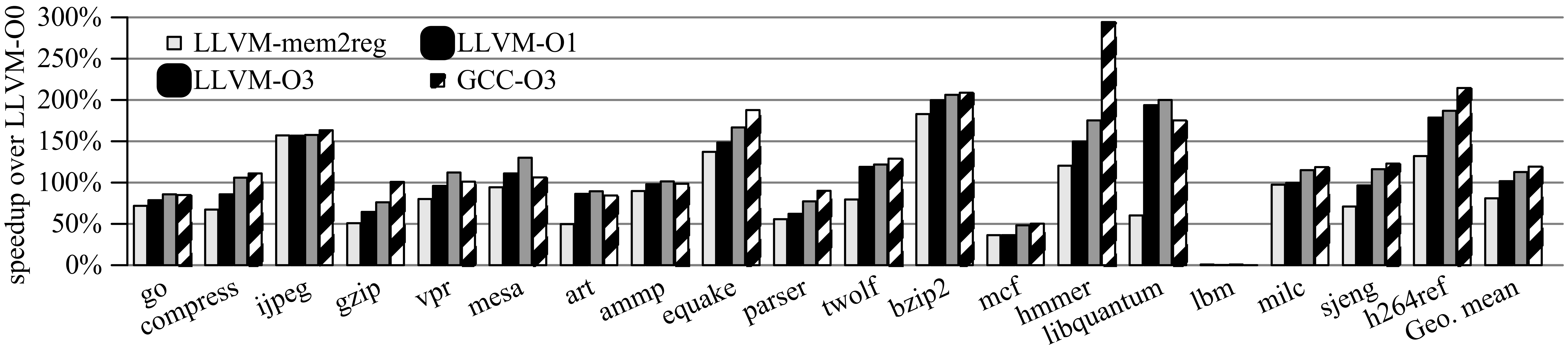
```
for (int i=0; i<n; i++) { S }
```

```
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 ⇒ 100% speedup

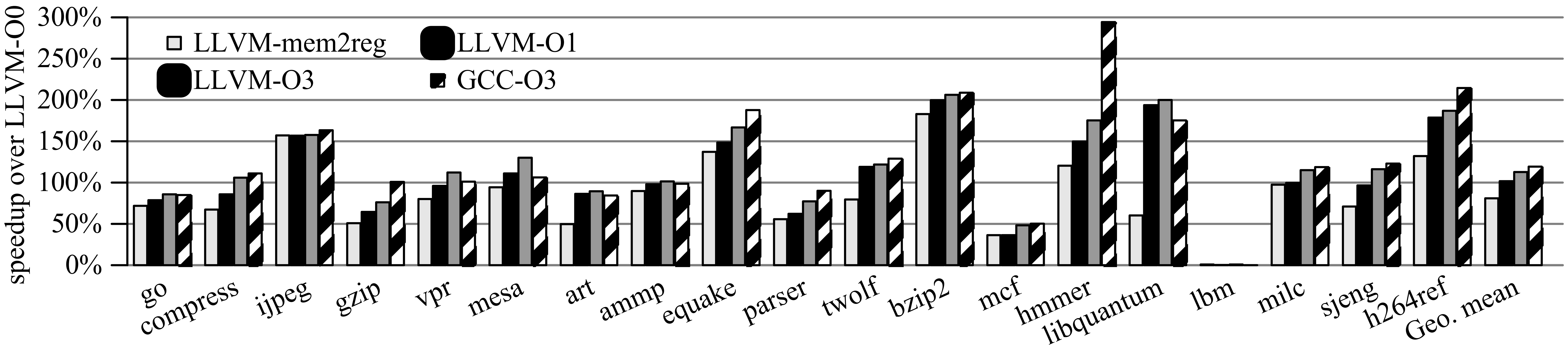
Example:

base time = 1.2s

optimized time = 0.87s ⇒ 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?

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

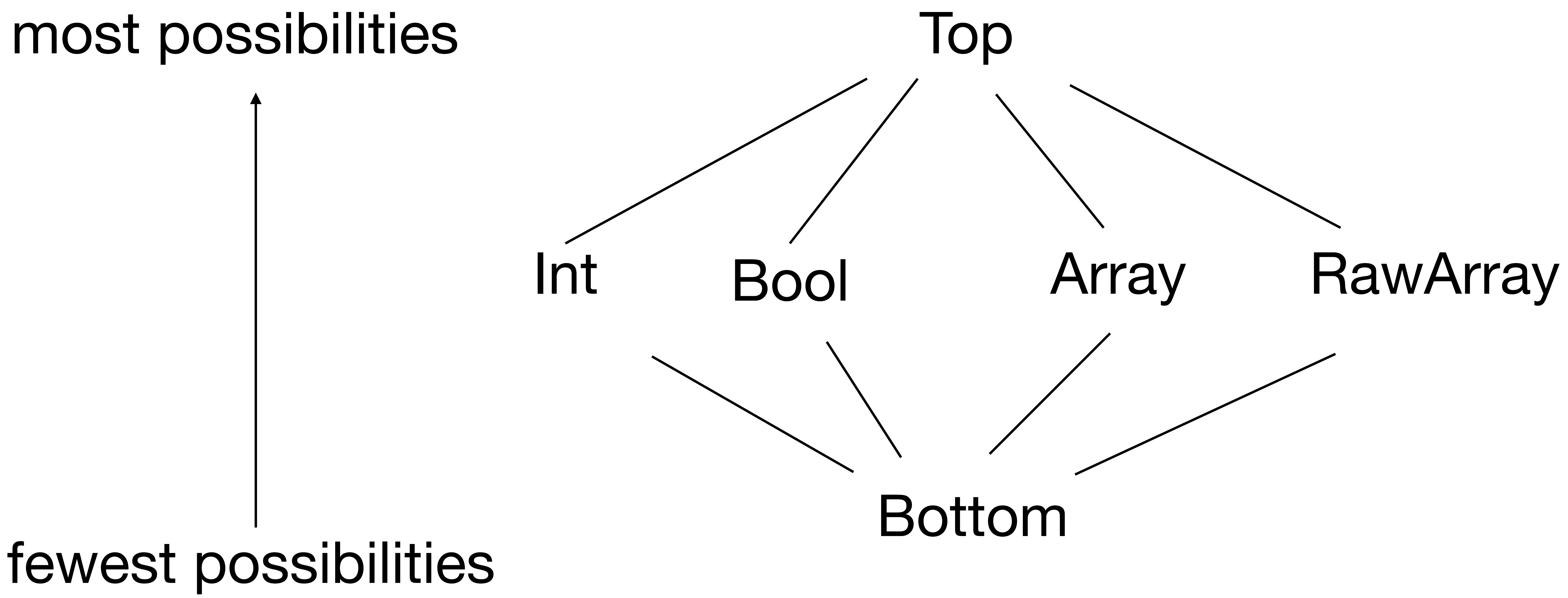
- ```
  x = g()
  assertInt(x)
  y = x + 2
  assertInt(y)
  assertInt(x)
  y2 = y >> 1
  z = y2 * x
  ...
  ```
- Which assertions can we remove?

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 Int
  - 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}
```