EECS 483: Compiler Construction

Lecture 18:
Optimization and Dataflow Analysis

March 23
Winter Semester 2026

Slides adapted from Steve Zdancewic

Announcements

- Exam Grading to be finished next week
- Assignment 4 due next Friday, April 4

Learning Objectives

Examples of optimizations that are commonly performed in compilers

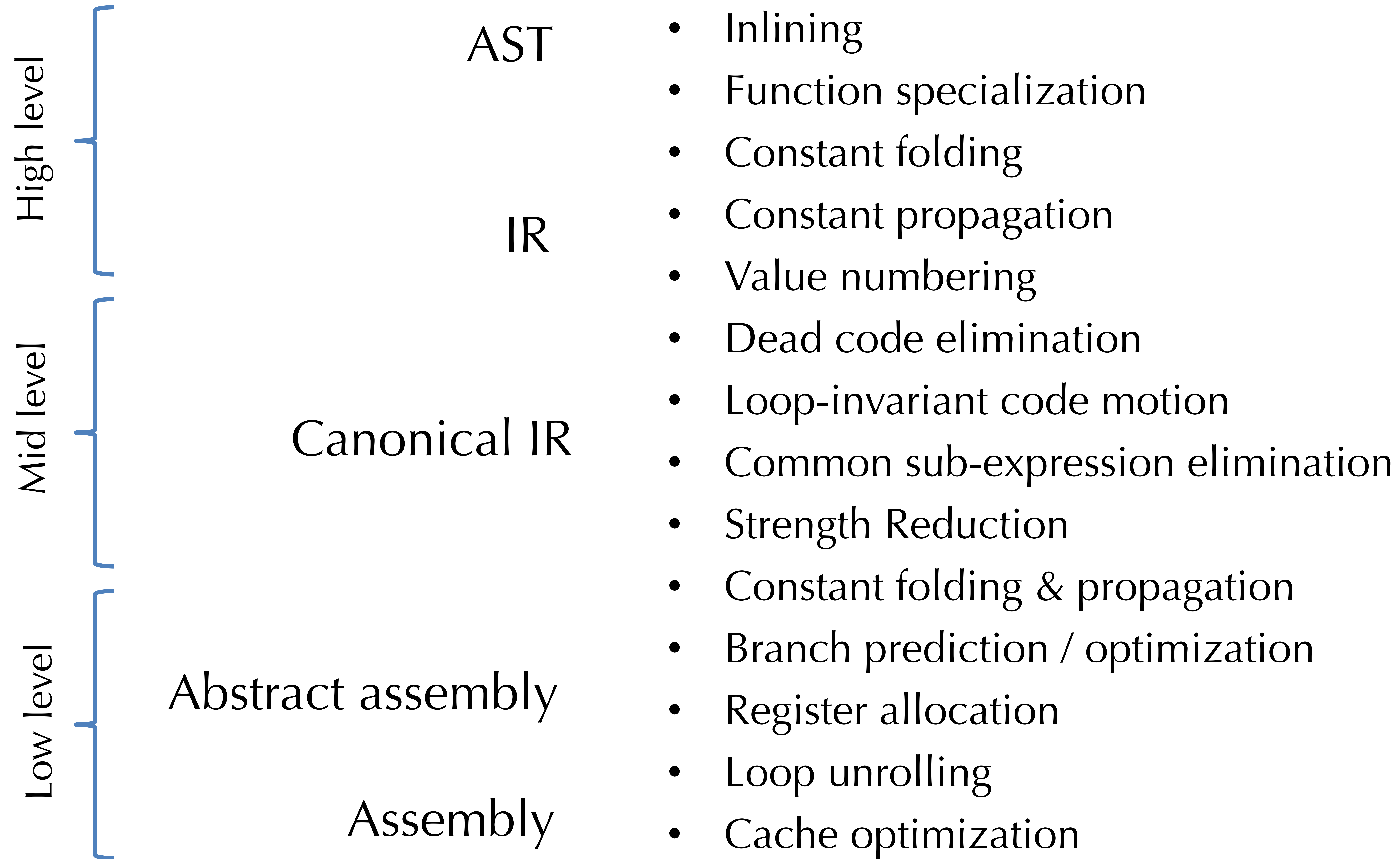
- at various stages, AST, SSA IR, Backend

Which optimizations are the most beneficial?

Another program analysis/optimization for Snake: Possible values and assertion removal.

OPTIMIZATIONS, GENERALLY

When to apply 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?

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;   (struck out)
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

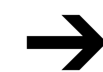
- Basic blocks not reachable by any trace leading from the starting basic block are *unreachable* and can be deleted.
  - Performed at the IR or assembly level
  - Improves cache, TLB performance

- 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.

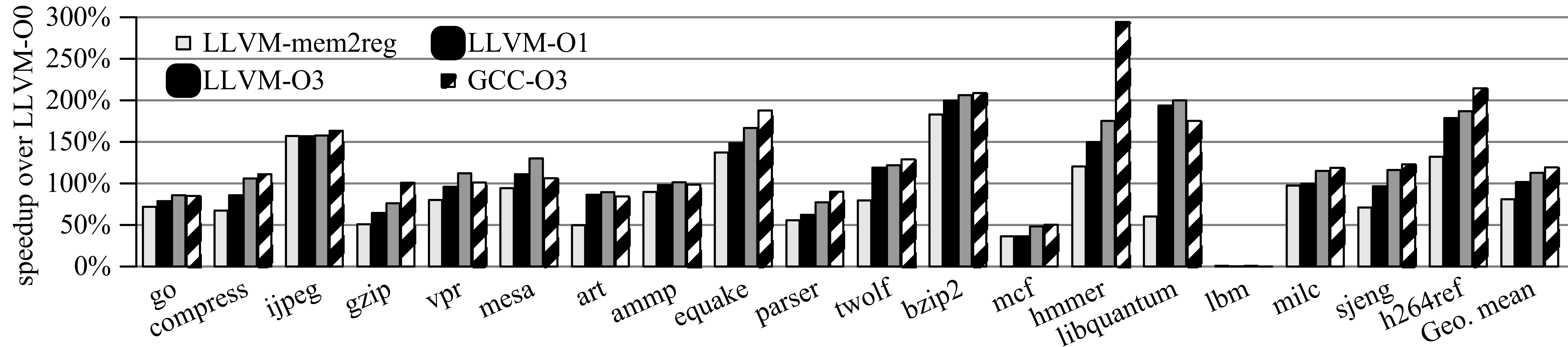
```
for (int i=0; i<n; i++) { S }
```

```
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

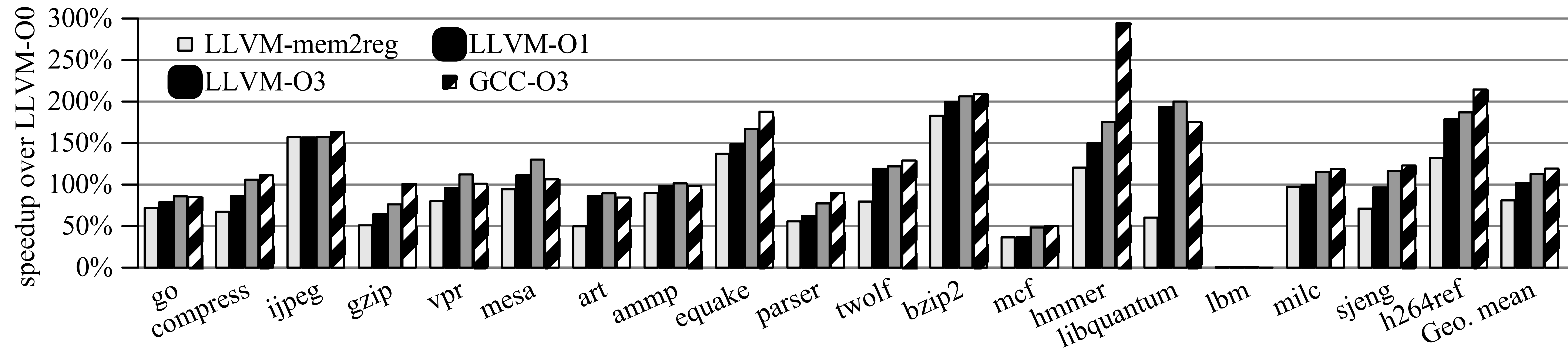
Example:

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

Assertion Removal

- Dynamic typing adds many runtime assertions into our program.

- ```
  let x = f() 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?

Assertion Removal

- Dynamic typing adds many runtime assertions into our program.

- ```
  let x = f() 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?**

Assertion Removal

- When is it correct to remove an assertion from our SSA program?

```
...
assertInt(x)
...
```

When we are **sure** that the assertion will succeed

In this case, if we are **sure** that x can only ever be a (tagged) integer at runtime.

- Appropriate analysis: determine what possible values x can take at runtime.

Possible Values Analysis

- To perform assertion removal, we need to figure out what possible values variables take at runtime.
  - Perform an analysis that says at every program point, the set of possible values that every variable might have at that point in the program.
  - Remove assertions that always would succeed on the possible values
- Rice's theorem applies: it's impossible to compute the exact correct sets. So we must approximate.

  Which way should we approximate?
  - Underapproximate: produce a **subset** of the true possible values. But might miss some
  - Overapproximate: produce a **superset** of the true possible values. But might include some that never happen
- For assertion removal, we need to **overapproximate**.
  - If our set is a superset of the true possible values, and still contains only tagged integers, then at runtime the possible value is definitely a tagged integer.
  - Might miss out on some assertion removals, but that's unavoidable.

Possible Values Analysis

- What do we mean by "possible values"?
- Performing this analysis at the SSA level. In SSA, a value is a 64-bit integer.
  - So we can represent a set of possible SSA values as a HashSet<i64> in Rust.

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

Problem: after x = f(), assuming f is an extern function, x may take on any value. That would be a huge set.
In general, a set of i64 values would take 2^(2^64) bits to represent!

Abstract Interpretation

- To keep space manageable, we need a different representation of sets, one that takes much less space than 2^(2^64) bits.
- This **inherently** means we are missing out on precision! But most of those sets are never going to come up in our analysis anyway.
- We design an "abstract domain" of possible value sets that is good enough to perform our analysis.
- To start, let's just worry about removing assertInt.
  - A simple abstract domain is to have just three elements:
    - Any (aka Top): this represents the set of all posssible 64-bit integers
    - Even (aka TaggedInt): this represents the even 64-bit integers
    - None aka Empty aka Bottom: the represents the empty set

Flow Functions

- At each program point, for each variable associate an approximation of what the possible values are.
- For each instruction, define a "flow function" that says how the possible values of variables change when an operation is performed

Flow Functions

- At each program point, for each variable associate an approximation of what the possible values are.
- For each instruction, define a "flow function" that says how the possible values are affected by performing the operation

- `x = y + z`

  - What is the **most precise information** we know about the possible values of x based on the possible values of y and z?

  - Poss(x) = Flow[+](Poss(y), Poss(z)) ~~ { y + z | y in Poss(y), z in Poss(z) }
    Flow[+](Any, Any) = Any
    Flow[+](Any, Even) = Flow[+](Even, Any) = Any
    Flow[+](Even, Even) = Even
    Flow[+](None, Q) = None
    Flow[+](P, None) = None

- Why is this correct? We output the most precise approximation of the set of all values that result from adding values in the input sets.
- None + Q = { y + z | y in EmptySet, z in Q } = EmptySet

Flow Functions

- At each program point, for each variable associate an approximation of what the possible values are.
- For each instruction, define a "flow function" that says how the possible values are affected by performing the operation

  - ```
    x = y << n where n >= 1
    ```

  - Poss(x) = Flow[<< n](Poss(y)) ~~ { y << n | y in Poss(y) }
    Flow[<< n](Any) = Even
    Flow[<<n](Even) = Even
    Flow[<<n](None) = None

- Note here that the case for Even **loses** precision:
  - { y << 1 | y in Even } = Multiples of 4 subset Even
    ```

Flow Functions

- At each program point, for each variable associate an approximation of what the possible values are.
- For each instruction, define a "flow function" that says how the possible values are affected by performing the operation

```
x = y * z
```

What is the **most precise information** we know about the possible values of x based on the possible values of y and z?

Poss(x) = Flow[*](Poss(y), Poss(z)) ~~ { y * z | y in Poss(y), z in Poss(z) }
Flow[+](Any, Any) = Any
Flow[+](Any, Even) = Flow[+](Even, Any) = Even
Flow[+](Even, Even) = Even
Flow[+](None, Q) = None
Flow[+](P, None) = None

Flow Functions

- At each program point, for each variable associate an approximation of what the possible values are.
- For each instruction, define a "flow function" that says how the possible values are affected by performing the operation

```
assertInt(x)
```

Poss(x) = Flow[assertInt](Poss(x))
Flow[assertInt](Any) = Even
Flow[assertInt](Even) = Even
Flow[assertInt](None) = None

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: Any}
2: {x: Even}
3: {x: Even, y: Even}
4: {x: Even, y: Even}
5: {x: Even, y: Even}
6: {x: Even, y: Even, y2: Any}
7: {x: Even, y: Even, y2: Any, z: Even}
```

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: Any}
2: {x: Even}
3: {x: Even, y: Even}
4: {x: Even, y: Even}
5: {x: Even, y: Even}
6: {x: Even, y: Even, y2: Any}
7: {x: Even, y: Even, y2: Any, z: Even}
```

Tag-checking Analysis

- At each program point, for each variable associate an approximation of what the possible values are.
- For each instruction, update that information accordingly
  To do a complete analysis: extend this to all SSA operations

- What about **blocks** and **functions?**

```
f(x):
  assertInt(x)
  assertInt(y)
  ...
```

  - What info do we have about x? about y?
    Collect the info from all the places that **branch to f,** taking a "union"
    We call these the **predecessors of f**, because they are the incoming edges of the control-flow graph.

  - Because we can have loops, **f** can be a predecessor of itself, so we have a similar circularity that we did in liveness.
    Same solution: initialize the information to be minimal (bottom in this case) and update iteratively

  - For functions, the predeccessors are places that **call f**.
    For main, there is a special implicit predecessor which is the entry point. This sets the input variable to Any because the program input is an array.

Loop Example

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

```
main(y):
  loop(i,a):
    thn():
      ret a
    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)
```

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

Initialize the blocks: main entry point arguments are Any, other blocks everything is None.

```
0: {y:Any}
1: {y,i,a:None}
2:
3: {y,i,a:None}
4: {y,i,a:None}
5:
6:
7:
8:
9:
10:
```

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

Next: apply the flow functions to update the internal nodes

```
 0: {y:Any}
 1: {y,i,a:None}
 2: {y,i,a,b:None}
 3: {y,i,a:None}
 4: {y,i,a:None}
 5: {y,i,a:None}
 6: {y,i,a,i':None}
 7: {y,i,a,i':None,x:Any}
 8: {y,i,a,i':None,x:Even}
 9: {y,i,a,i',a':None,x:Even}
10: {y,i,a,i',a':None,x:Even}
```

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

To start a new iteration, initialize blocks based on the previous round's info about predecessors

Previous Round	Next Round
0: {y:Any}	0: ?
1: {y,i,a:None}	1: ?
2: {y,i,a,b:None}	2:
3: {y,i,a:None}	3: ?
4: {y,i,a:None}	4: ?
5: {y,i,a:None}	5:
6: {y,i,a,i':None}	6:
7: {y,i,a,i':None,x:Any}	7:
8: {y,i,a,i':None,x:Even}	8:
9: {y,i,a,i',a':None,x:Even}	9:
10: {y,i,a,i',a':None,x:Even}	10:

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

To start a new iteration, initialize blocks based on the previous round's info about predecessors

Previous Round

```
0: {y:Any}
1: {y,i,a:None}
2: {y,i,a,b:None}
3: {y,i,a:None}
4: {y,i,a:None}
5: {y,i,a:None}
6: {y,i,a,i':None}
7: {y,i,a,i':None,x:Any}
8: {y,i,a,i':None,x:Even}
9: {y,i,a,i',a':None,x:Even}
10: {y,i,a,i',a':None,x:Even}
```

Next Round

```
0: {y:Any}
1: {y:Any U None,
    i:Any U None,
    a: Even U None}
2:
3: ?
4: ?
5:
6:
7:
8:
9:
10:
```

the loop(i,a) body 1 has two predecessors:

br loop(y, 0)

br loop(i', a')

Take the "union" of their information

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

To start a new iteration, initialize blocks based on the previous round's info about predecessors

Previous Round	Next Round
0: {y:Any}	0: {y:Any}
1: {y,i,a:None}	1: {y:Any,i:Any,a:Even}
2: {y,i,a,b:None}	2:
3: {y,i,a:None}	3: ?
4: {y,i,a:None}	4: ?
5: {y,i,a:None}	5:
6: {y,i,a,i':None}	6:
7: {y,i,a,i':None,x:Any}	7:
8: {y,i,a,i':None,x:Even}	8:
9: {y,i,a,i',a':None,x:Even}	9:
10: {y,i,a,i',a':None,x:Even}	10:

the loop(i,a) body 1 has two predecessors:
    br loop(y, 0)
    br loop(i', a')
Take the "union" of their information

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

To start a new iteration, initialize blocks based on the previous round's info about predecessors

Previous Round	Next Round
0: {y:Any}	0: {y:Any}
1: {y,i,a:None}	1: {y:Any,i:Any,a:Even}
2: {y,i,a,b:None}	2:
3: {y,i,a:None}	3: {y,i,a,b:None}
4: {y,i,a:None}	4: {y,i,a,b:None}
5: {y,i,a:None}	5:
6: {y,i,a,i':None}	6:
7: {y,i,a,i':None,x:Any}	7:
8: {y,i,a,i':None,x:Even}	8:
9: {y,i,a,i',a':None,x:Even}	9:
10: {y,i,a,i',a':None,x:Even}	10:

thn() and els() each have one predecessor

cbr b thn() els()

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

Update the rest of the internal nodes

Current Round

```
0: {y:Any}
1: {y:Any,i:Any,a:Even}
2:
3: {y,i,a,b:None}
4: {y,i,a,b:None}
5:
6:
7:
8:
9:
10:
```

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

Update the rest of the internal nodes

Current Round

```
 0: {y:Any}
 1: {y:Any,i:Any,a:Even}
 2: {y:Any,i:Any,a:Even,b:Any}
 3: {y,i,a,b:None}
 4: {y,i,a,b:None}
 5: {y,i,a:None}
 6: {y,i,a,i':None}
 7: {y,i,a,i':None,x:Any}
 8: {y,i,a,i':None,x:Even}
 9: {y,i,a,i',a':None,x:Even}
10: {y,i,a,i',a':None,x:Even}
```

Since the results changed, we perform another iteration

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

Previous Round		Next Round	
0:	{y:Any}	0:	{y:Any}
1:	{y:Any,i:Any,a:Even}	1:	{y:Any,i:Any,a:Even}
2:	{y:Any,i:Any,a:Even,b:Any}	2:	
3:	{y,i,a,b:None}	3:	?
4:	{y,i,a,b:None}	4:	?
5:	{y,i,a:None}	5:	
6:	{y,i,a,i':None}	6:	
7:	{y,i,a,i':None,x:Any}	7:	
8:	{y,i,a,i':None,x:Even}	8:	
9:	{y,i,a,i',a':None,x:Even}	9:	
10:	{y,i,a,i',a':None,x:Even}	10:	

the loop(i,a) body 1 has two predecessors:

    br loop(y, 0)
    br loop(i', a')

Take the "union" of their information

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

	Previous Round		Next Round
0:	{y:Any}	0:	{y:Any}
1:	{y:Any,i:Any,a:Even}	1:	{y:Any,i:Any,a:Even}
2:	{y:Any,i:Any,a:Even,b:Any}	2:	
3:	{y,i,a,b:None}	3:	{y:Any,i:Any,a:Even,b:Any
4:	{y,i,a,b:None}	4:	{y:Any,i:Any,a:Even,b:Any
5:	{y,i,a:None}	5:	
6:	{y,i,a,i':None}	6:	
7:	{y,i,a,i':None,x:Any}	7:	
8:	{y,i,a,i':None,x:Even}	8:	
9:	{y,i,a,i',a':None,x:Even}	9:	
10:	{y,i,a,i',a':None,x:Even}	10:	

thn() and els() each have one predecessor
cbr b thn() els()

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

Update the rest of the internal nodes

Current Round

```
0: {y:Any}
1: {y:Any,i:Any,a:Even}
2:
3: {y:Any,i:Any,a:Even,b:Any}
4: {y:Any,i:Any,a:Even,b:Any}
5:
6:
7:
8:
9:
10:
```

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

Update the rest of the internal nodes

Current Round

```
0: {y:Any}
1: {y:Any,i:Any,a:Even}
2: {y:Any,i:Any,a:Even,b:Any}
3: {y:Any,i:Any,a:Even,b:Any}
4: {y:Any,i:Any,a:Even,b:Any}
5: {y:Any,i:Even,a:Even,b:Any}
6: {y:Any,i:Even,a:Even,b:Any,i':Even}
7: {y:Any,i:Even,a:Even,b:Any,i':Even,x:Any}
8: {y:Any,i:Even,a:Even,b:Any,i':Even,x:Any}
9: {y:Any,i:Even,a:Even,b:Any,i':Even,x:Int}
10: {y:Any,i:Even,a:Even,b:Any,i':Even,x:Int,a':Int}
```

If we repeat one more iteration, we get the same result, and we have our final analysis

```
main(y):
  loop(i,a):
    thn():
     3
     ret a
    els():
     4assertInt(i)
     5i' = i - 2
     6x = g()
     7assertInt(a)   [struck through]
     8
     assertInt(x)
     9a' = a + x
    10br loop(i', a')
   1b = i == 0
   2cbr b thn() els()
 0
 br loop(y, 0)
```

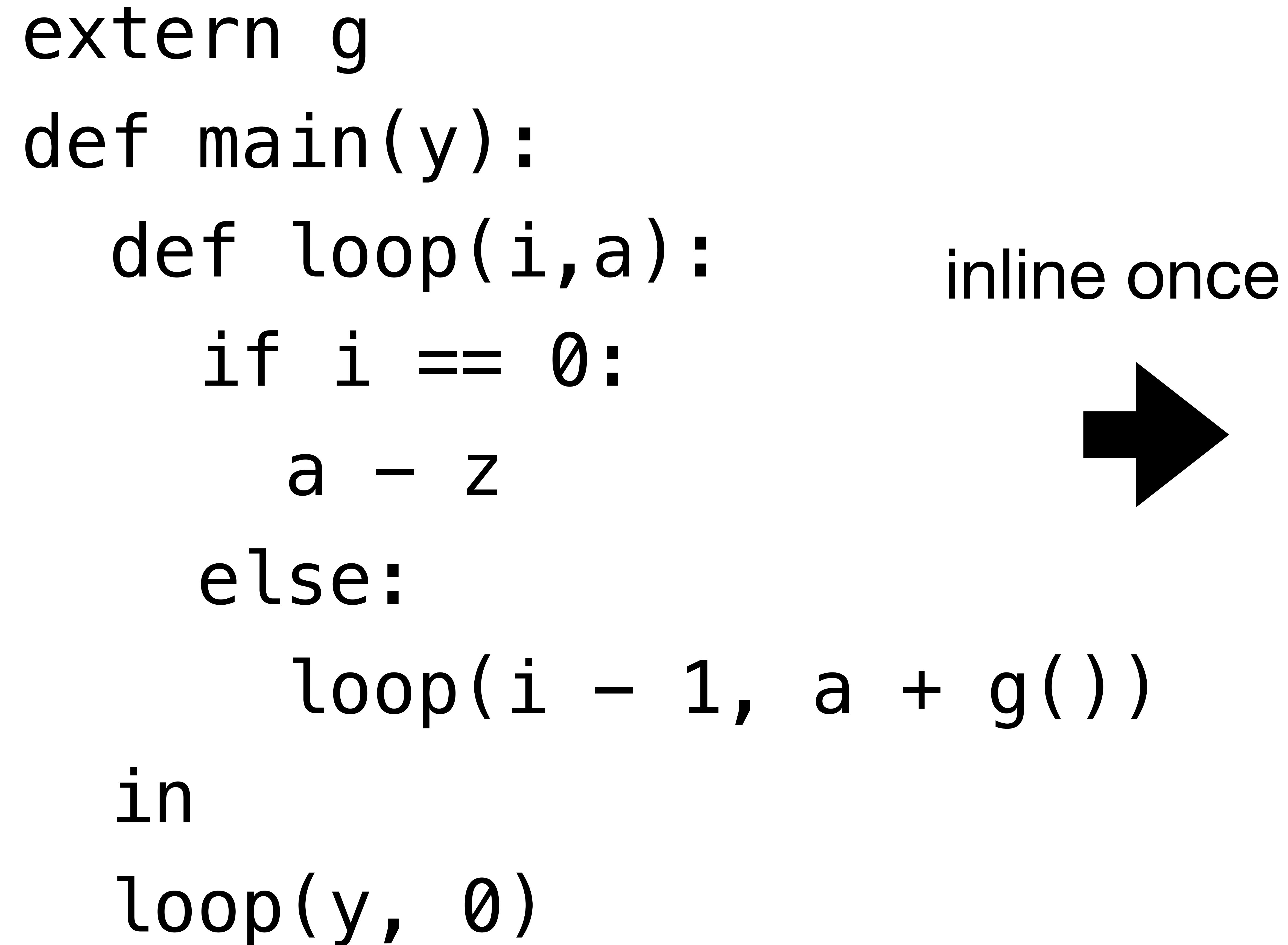
Update the rest of the internal nodes

Current Round

```
 0: {y:Any}
 1: {y:Any,i:Any,a:Even}
 2: {y:Any,i:Any,a:Even,b:Any}
 3: {y:Any,i:Any,a:Even,b:Any}
 4: {y:Any,i:Any,a:Even,b:Any}
 5: {y:Any,i:Even,a:Even,b:Any}
 6: {y:Any,i:Even,a:Even,b:Any,i':Even}
 7: {y:Any,i:Even,a:Even,b:Any,i':Even,x:Any}
 8: {y:Any,i:Even,a:Even,b:Any,i':Even,x:Any}
 9: {y:Any,i:Even,a:Even,b:Any,i':Even,x:Int}
10: {y:Any,i:Even,a:Even,b:Any,i':Even,x:Int,a':Int}
```

With all of this work, we can remove 1 assertion: assertInt(a)

We can't prove that assertInt(i) succeeds because the initial value of y might not be an integer...



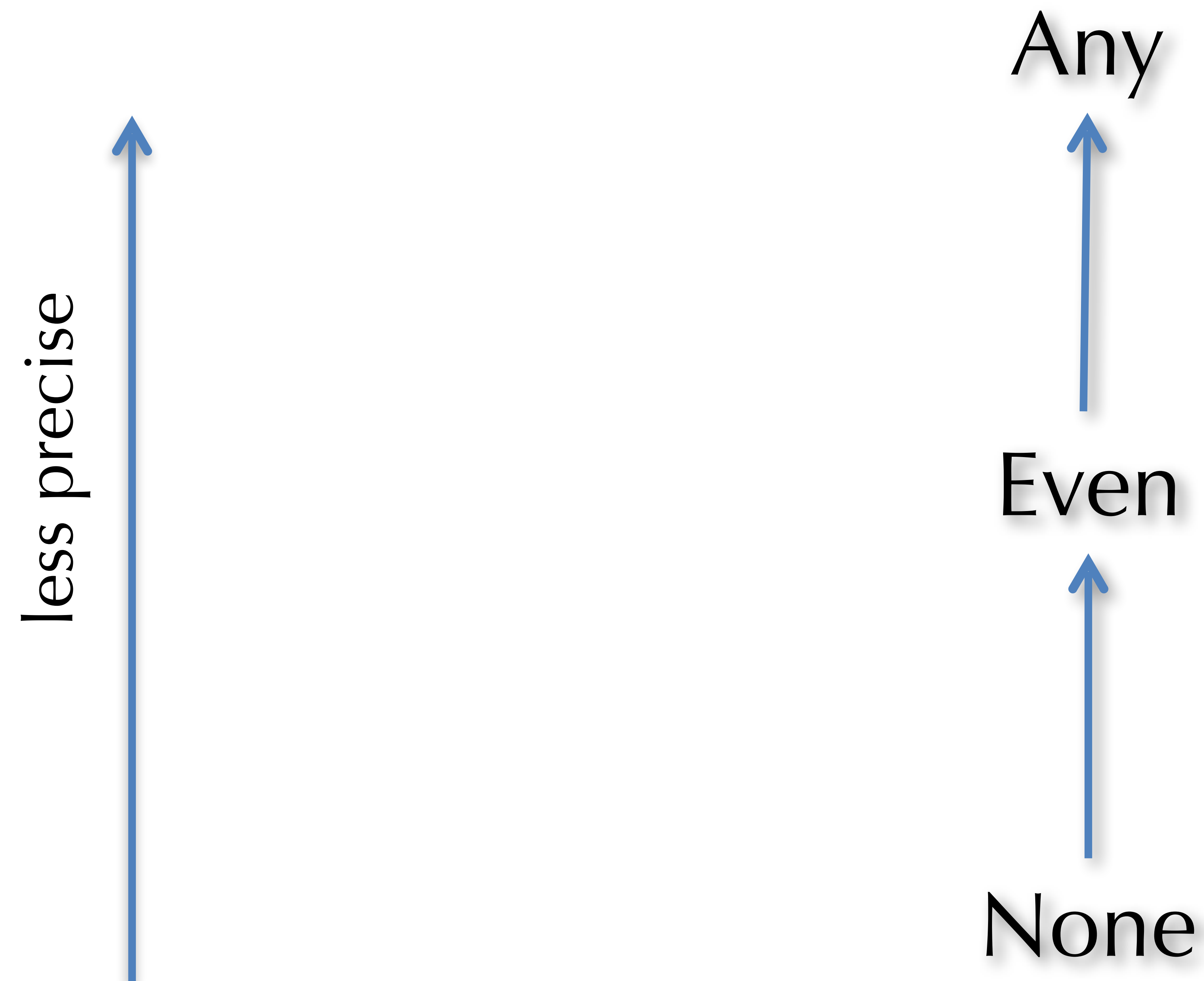
inline once

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

If we re-do the analysis, no i is always an Int

Abstract Interpretation

- We used a simple interpretation for just removing assertInt
  - A simple abstract domain is to have just three elements:
    - Any (aka Top): this represents the set of all posssible 64-bit integers
    - Even (aka TaggedInt): this represents the even 64-bit integers
    - None aka Empty aka Bottom: the represents the empty set



Abstract Interpretation

- We used a simple interpretation for just removing assertInt
  - What about for Booleans?
  - Update all flow functions accordingly
  - Similar for arrays
  - Tradeoff: more complex Abstraction means more precise analysis, but more space usage, more difficult to define