Compiler Notes

ThanhVu Nguyen

April 30, 2021

Contents

Chapter 1

Optimizations

- recall 5 phases of compiler: lexer, parser, (type checker, operational semantics), optimization, translate to target machine code (ASM)
- modern compilers: most actions happen in optimization phase

1.1 Intermediate Representation (IR)

- provides an intermediate level of abstraction
- has more details than the source code
  - Optimizations happen on the IR
- but less details than the target (machine, or assembly code) ...

1.1.1 Three-Address IR

- every instructions has the form

```
x := y op z    # binary , e.g., x:= y + z
x := op y      # unary  , e.g.,  y :=  -y
```

  - `y,z` are registers or constants
- compound expression `x + y * z` is translated to:

```
t1 := y * z
t2 := x + t1
```

1.2 Optimization Overview

- Largest (most complicated) phase of compiler
- Where to perform optimization
  - On AST:
    - Pro: Machine independent
    - Con: Too high level (cannot too too much optimization)

- On ASM:
  * Pro: low level, expose many details and optimization opportunies
  * Con: machine dependent, reimplement the optimization if switch to a different target
- On IR:
  * Pro: machine independent
  * Pro: low level enough to expose optimization opportunties

1.2.1 3-address code

```
P -> S P | S
S -> id := id op id    #op are things like +, -, * ...
 |   id := op id
 |   id := id
 |   push id
 |   id := pop
 |   if id relop id goto L   # relop < = > ...
 |   L:
 |   jump L
```

1.2.2 Basic Block

A **basic block** is a maximal sequence of instructions with

- no label (except in the first instruction)
- no jump (except in the last instruction)

```
L:
....
....
....
jump M
```

- cannot jump into a middle of a block (except at the beginning)
- cannot jump out of a middlew of a block (except at the end)
- Consider this basic block

```
L:                     (1)
t := 2 * x             (2)
w := t + x             (3)
if w > 0 goto to L'    (4)
```

- Because (3) executes AFTER (2), we can
  - change (3) to `w := 3 * x`
  - remove 2 (assuming `t` is not used anywhere else)

1.2.3 Control-Flow Graph

- A **Control-Flow graph** (CFG) is a directed graph
  - basic blocks are nodes
  - edge from a block A to a block if the execution can pass from the last instruction in A to the first instruction in B
    * E.g., the last instruction is 'jump Lb'
  - We can represent the body of a method (or function or procedure) as a CFG
- Goals of optimization
  - Minimize Execution time (most often)
  - Minimize Code size (e.g., embedded system)
  - Minimize Operations to Disks (e.g., Database)
  - Minimize Power Consumption (e.g., sensor, smart phones, watches)
  - Important: Need to preserve the semantics of the program
    * whatever results we get from the original one, we need to get the same results in the optimization version

1.2.4 3 granularity levels of optimizations

1. Local optimizations: Apply optimization to basic block in isolation
2. Global optimizations: Apply optimization to the CFG in isolation
3. Inter-procedural optimizations: Apply optimization to the entire program (consists multiple methods and functions)

- Most compilers do (1: local), many do (2: global), very few do (3)
- In practice, people DO NOT use the fanciest/most optimized algorithms
  - They have low pay-offs
  - Too hard/complex to implement (this might affect correctness preservation)
  - Their analyses too expensive during compilation time
  - **Goal**: maximum benefit for minimum cost

1.3 Local Optimization: (optimization applied to basic block)

1.3.1 Algebraic Simplifications

- can delete some statements

```
x := x + 0    # or x:=  0 + x
x := x * 1    # or x:= 1 * x
```

- can simplify some statements

```
x := x * 0   =>   x := 0
x := y ** 2  =>   x := y * y   # make call to library (expensive operation),
                               # usually has loop to do exp
x := x * 8   =>   x := x << 3   # on some machines << is faster than *modern computers)
x := x * 15  =>   x := x << 4; x := t - x
```

1.3.2 Constant Folding

- operations on constants can be computed at compile time
  - if there is a statement `x := y op z`, where y and z are constants, then `y op z` can be computed at compile time
  - Example:

```
x : = 2 + 3   => x := 5
if 2 > 0: code   => if true: code   => code
if 2 < 0: code   => if false: code =>   code never get executed
```

- Constant folding can be dangerous (gives different results)
  - Compile program on machine X
  - Run the compiled program on machine Y
  - X and Y might have diff architectures
    - `a := 1.5 + 3.7 => a := 5.2 on X`
    - `a := 1.5 + 3.7 => a:   5.1999 on Y`
    - `a = "1.5 + 3.7"`

1.3.3 Unreachable Examples

- debug macro

```
#define DEBUG 0
if (DEBUG) then ....
```

- libraries (not everything in the library are used)

1.3.4 Single Assignment form

- each register (id) occurs only ONCE on the left-hand side of an assignment

```
x := z + y       =>     b := z + y
a := x           =>     a := b
x := 2 * x       =>     x := 2 * b
```

- converting to SA could be tricky in many code regions (e.g., within loops)

1.3.5 Optimizations on SA blocks

Common Subexpression Elimination

- if a basic block is in SA form
- a definition `x :=` is the first use of `x` in a block
- then when 2 assignments have the same rhs, then they compute the same value

```
x := y + z              =>   x:= y + z
...                     =>   ...
w := y + z              =>   w:= x
```

Copy Propagation

```
b := z + y   =>    b := z + y
a := b       =>    a := b
x := 2 * a   =>    x := 2 * b
```

- only useful for enabling other optimizations
  - eliminate dead code
  - constant folding
- Example

```
a := 5                  a := 5
x := 2 * a     ===>     x := 10
y := x + 6              y := 16
t := x * y              t := 160
```

Dead code elimination

- if `w:=rhs` appears in a basic block and `w` does not appear anywhere else, then `w:=rhs` is dead and can be removed

Summary for local optimization

- each local optimization does little thing by itself
- but they interact (performing an optimization enables another)
- compiler: repeat optimization until no other improvement is possible
  - but usually compilers has heuristics to determine when to stop

Inclass Example

```
# initial
a := x ** 2
b := 3
c := x
d := c * c
e := b * 2
f := a + d
g := e * f

# final version
a := x * x
g := 12 * a
....
```

Peephole Optimization

- **Peephole**: is a short sequence of (usually contiguous) instructions
- The compiler replaces that peephole (sequence) with another one that is equivalent (but faster)
  - `i1,...,in  ->  j1,...,jm`
- Peephole is often performed on assembly code
- Examples

```
# 1
a := b          =>  a := b
b := a

# 2
a := a + 1      => a:= a + 3
a := a + 2
```

- Just like local optimization, peephole opt must be applied repeatedly for maximum effect
- "Optimization" is misnamed
  - Compiler does not produce an "optimal" version
  - it only attempts to improve the code by repeatedly applying various optimization techniques

1.4 Global Optimization

1.4.1 Dataflow Analysis

```
              x:=3
              B > 0
             /     \
            /       \
      Y:= Z + W      Y:=0
            \       /
             \     /
             A := 2 * x   #  can replace x with 3
```

- To replace a use of a variable x by a constant k, we need to ensure that
  - on **every path** to the use of x, the last assignment to x has the form

```
x := k
```

- dataflow analysis (global)
  - an analysis of the entire control flow graph

```
              x:=3
              B > 0
             /     \
            /       \
        Y:= Z + W     Y:=0
          x:=4       /
             \      /
            A := 2 * x   # cannot replace x with anything
```

- Global optimization tasks (e.g., dataflow anlaysis) have shared traits
  - to make some optimization at a location `X`, then we need to know the properties at `X` (we need to know the invariant properties at `X`)
  - requires knowledge of the *entire* program
  - it's OK to be *conservative*. If the compiler doesn't know what is true, then it will say it doesn't know.
    - always safe to say it doesn't know.

1.5 Constant Propagation

- To replace a use of a variable `x` by a constant `k`, we need to ensure that
  - on **every path** to the use of x, the last assignment to x has the form `x := k`
- The property that we are interested in is checking if `x := k` (at some location `L`)?
- 3 Values that the analysis can give at location `L` about the property `x:=k`
  - **BOTTOM** $\bot$: this location is NOT reachable
  - **k**: `x == k`
  - **TOP** $\top$: no idea what `x` could be here

Example

```
              [x = TOP]
              x:=3
              [x == 3]
              B > 0
              [x == 3 ]
             /     \
            /       \
   Y:= Z + W       Y:=0
    [x==3]           [x ==3]
     x:=4
    [x==4]          /
            \      /
            [x==TOP]
            A := 2 * x
```

- Given global constant information, it is easy to perform the optimization

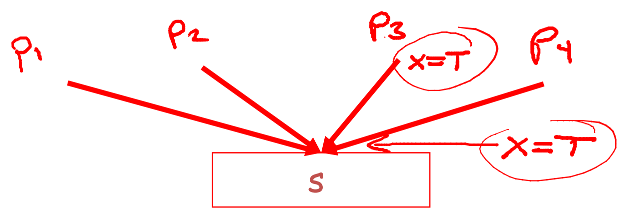
- Simply inspect the `x = ?` associated with a statement using `x`
- If `x` is constant at that point replace that use of x by the constant

- But how do we compute the properties `x = ?`

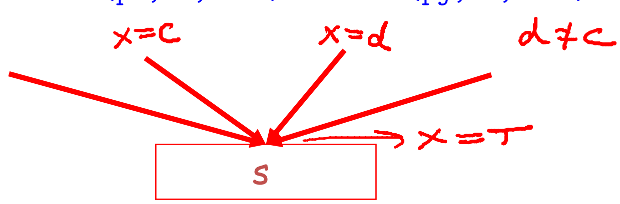
Constant Propagation Rules

- The analysis of a complicated program can be expressed as a combination of *simple rules* relating the *change in information* between adjacent statements.
- Idea: "push" or "transfer" information from one stmt to the next
  - For each stmt `s`, compute information about the value of x before and after `s`
  - `C(s,x,in)` = value of `x` before `s`
  - `C(s,x,out)` = value of `x` after `s`
- Define transfer functions (rules) that transfer information one statement to another
  - In the following rules, let statement `s` have immediate predecessor statements `p1,...,pn`
  - Rules **1-4** defined below relate the out of one statement to the in of the next statement
  - Rules **5-8** defined below relate the in of a statement to the out of the same statement

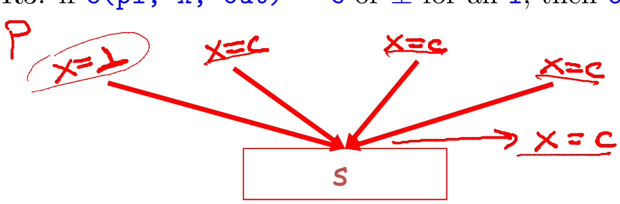
1. R1: if `C(pi, x, out) = S` for any `i`, then `C(s, x, in) = S`



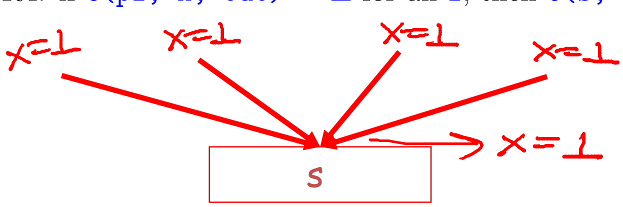
2. R2: if `C(pi, x, out) = c & C(pj, x, out) = d & d <> c` then C(s, x, in) = S



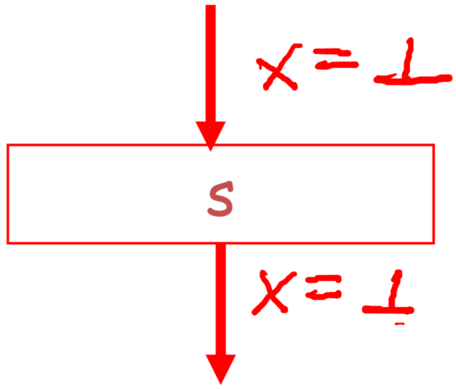
3. R3: if `C(pi, x, out) = c` or ⊥ for all `i`, then `C(s, x, in) = c`



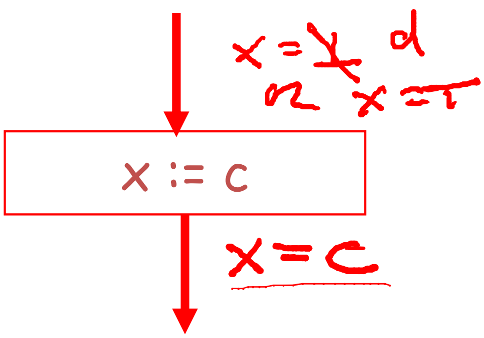
4. R4: if `C(pi, x, out) =` ⊥ for all `i`, then `C(s, x, in) =` ⊥



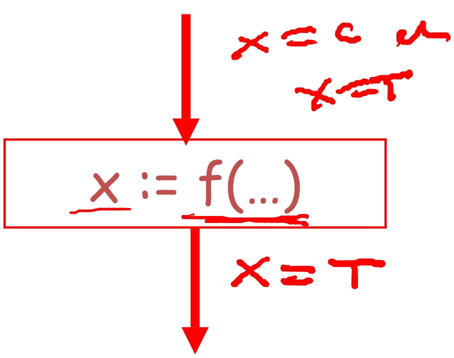
5. R5: `C(s, x, out) = ⊥` if `C(s, x, in) = ⊥`



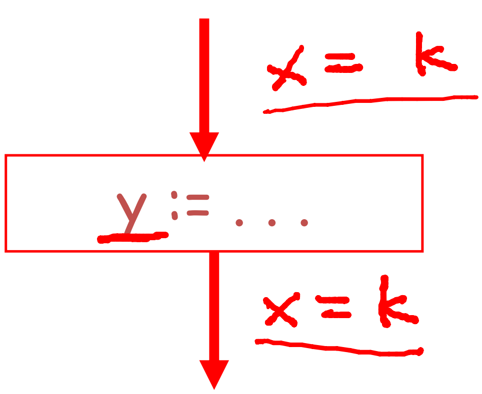
6. R6: `C(x := c, x, out) = c` if `c` is a constant



7. R7: `C(x:=f(...), x, out) = ⊤`

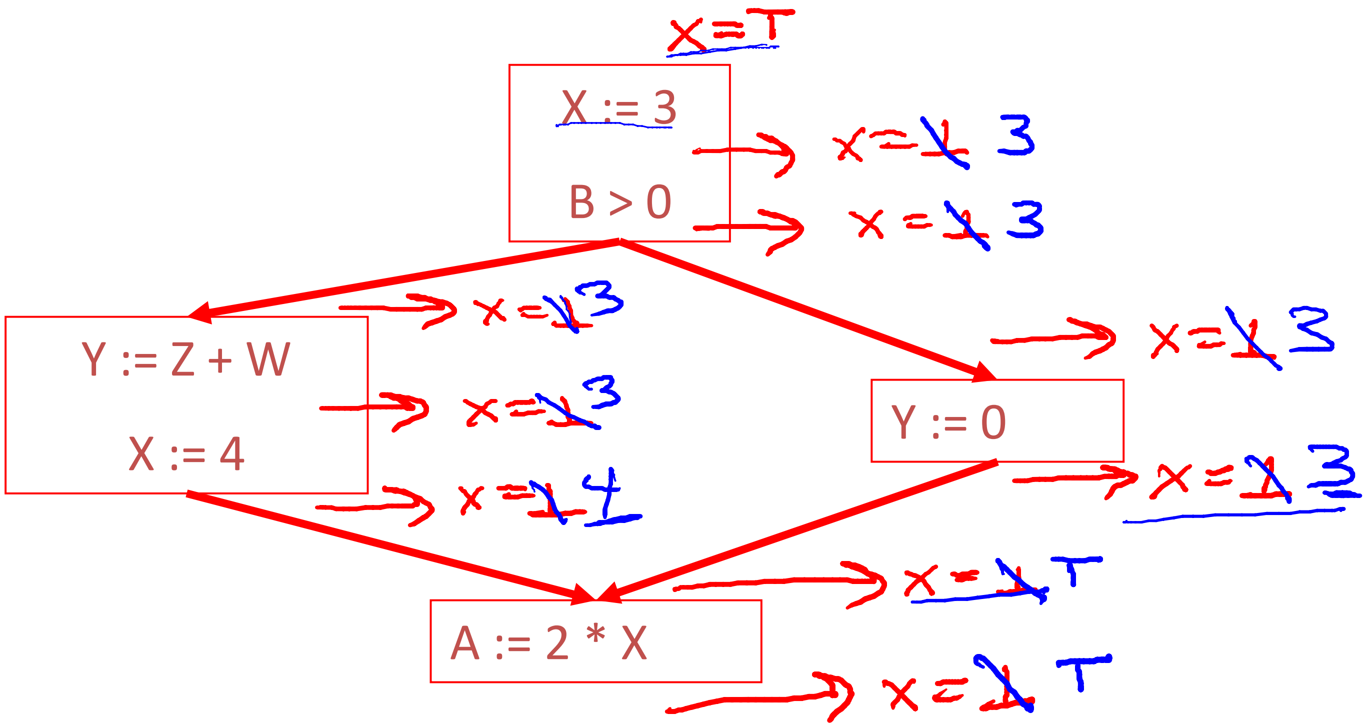


8. R8: `C(y:=...,x,out) = C(y:=...,x,in)` if `x <> y`



1.5.1 Algorithm

1. For every entry s to the program, set `C(s, x, in) = ⊤`
2. Set `C(s, x, in) = C(s, x, out) = ⊥` everywhere else
3. Repeat until all points satisfy rules 1-8:
   - Pick `s` not satisfying 1-8 and update using the appropriate rule



1.5.2 Orderings

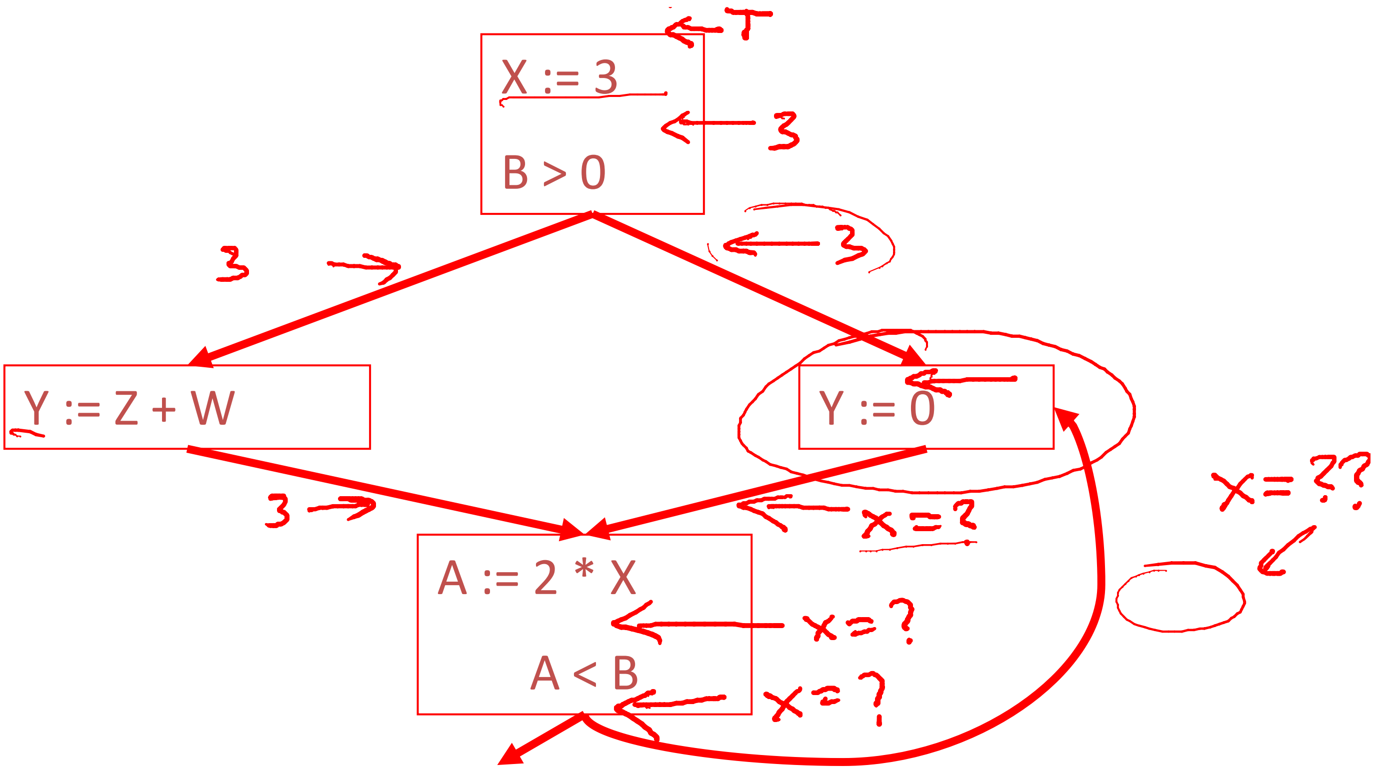
- We can simplify the presentation of the analysis by ordering the (abstract) values: $\bot < c < \top$
- $\top$ is the greatest value, $\bot$ is the least, and allconstants are in between and *incomparable*
- Let `lub` be the *least-upperbound* in this ordering
- Rules 1-4 can be written using `lub`:

```
C(s, x, in) = lub  C(p, x, out) | p is a predecessor of s
```

- Lub also explains why the algorithm terminates
  - Values start as $\bot$ and only increase
  - $\bot$ can change to a constant, and a constant to $\top$
  - Thus, `C(s, x, in/out)` can change at most *twice*
  - Thus the constant propagation algorithm is *linear* in (non-loop) program size

    Number of steps = Number of `C(...)` values computed *2 = Number of program statements * 4

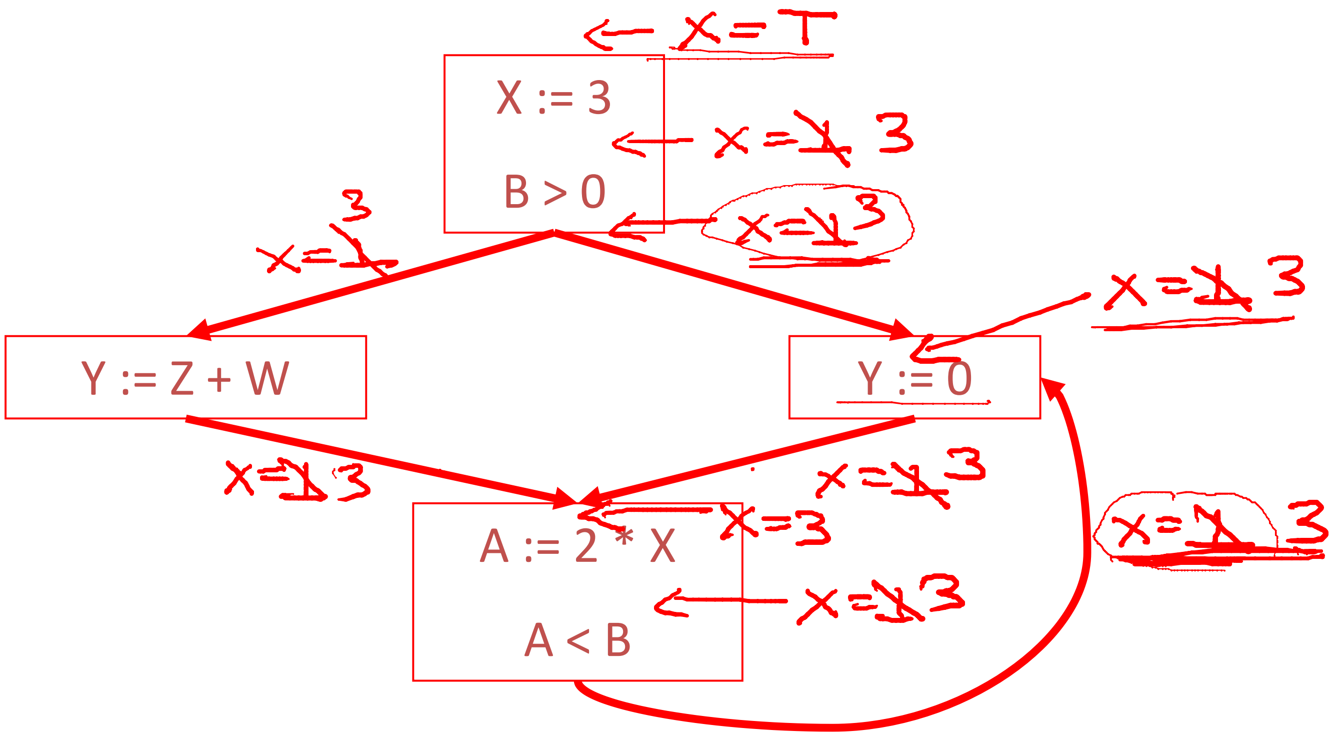
1.5.3 Loops



- Consider the statement `Y := 0`
- To compute whether `X` is constant at this point, we need to know whether `X` is constant at the two predecessors
  - `X := 3`
  - `A := 2 * X`
- Cycle: but the info for `A := 2 * X` depends on its predecessors, including `Y := 0`

Sol : Initialization of everything to ⊥ helps break the cycle

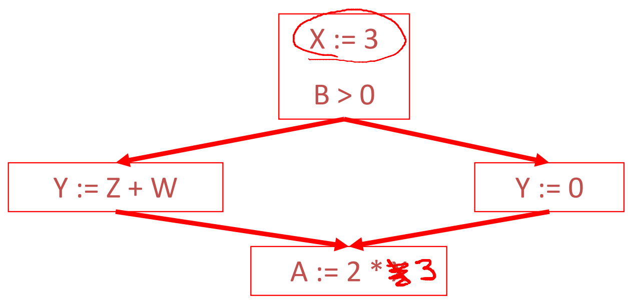
- Because of cycles, all points must have values at all times
- Intuitively, assigning some initial value allows the analysis to break cycles
- The initial value ⊥ means "So far as we know, control neeer reaches this point"



1.6 Liveness Analysis

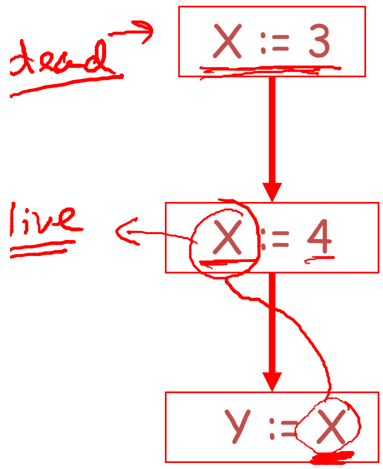
1.6.1 Definition

- Once constants have been globally propagated, we want to eliminate dead code



  - After constant propagation, X := 3 is dead (assuming X not used elsewhere)

- Example



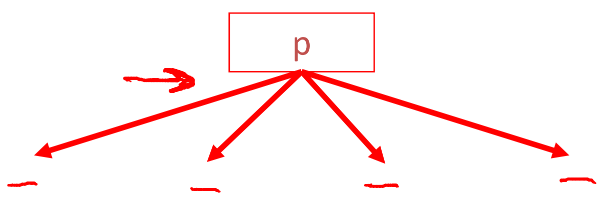
- The first value of `x` is *dead* (never used)
- The second value of `x` is *live* (may be used)

- Def: a variable `x` is *live* at statement `s` if
  - There exists a statement `s'` that uses `x`
  - There is a path from `s` to `s'` that has no *intervening assignment* to `x`
  - A statement `x:=...` is dead code if `x` is dead after the assignment
    * Dead statements can be deleted from the program

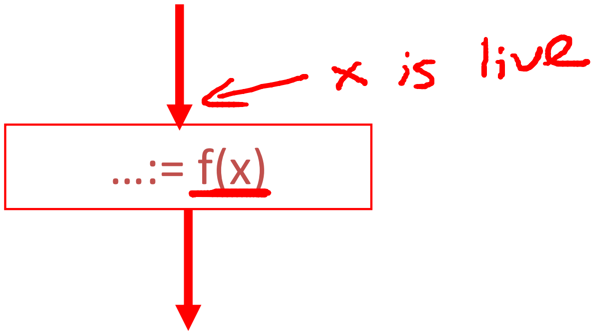
Liveness Rules

- We can express liveness in terms of *information transferred* between adjacent statements, just as in copy propagation
- Liveness is simpler than constant propagation, since it is a boolean property (true or false)
- Define transfer functions (rules) that transfer information one statement to another

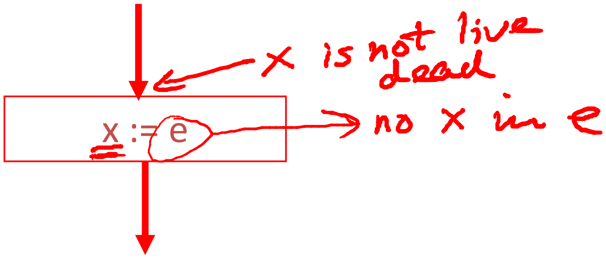
1. R1: `L(p, x, out) = ∨  L(s, x, in) | s a successor of p`



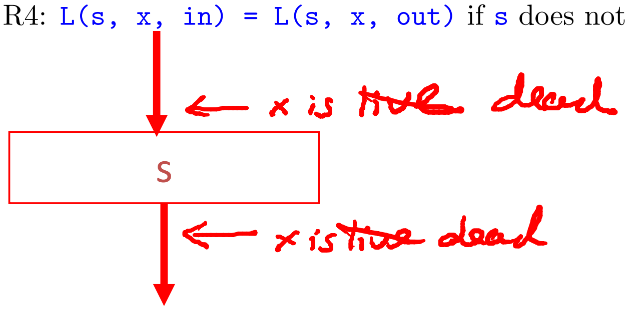
2. R2: `L(s, x, in) = true` if `s` refers to `x` on the rhs



3. R3: `L(x := e, x, in) = false` if `e` does not refer to `x`

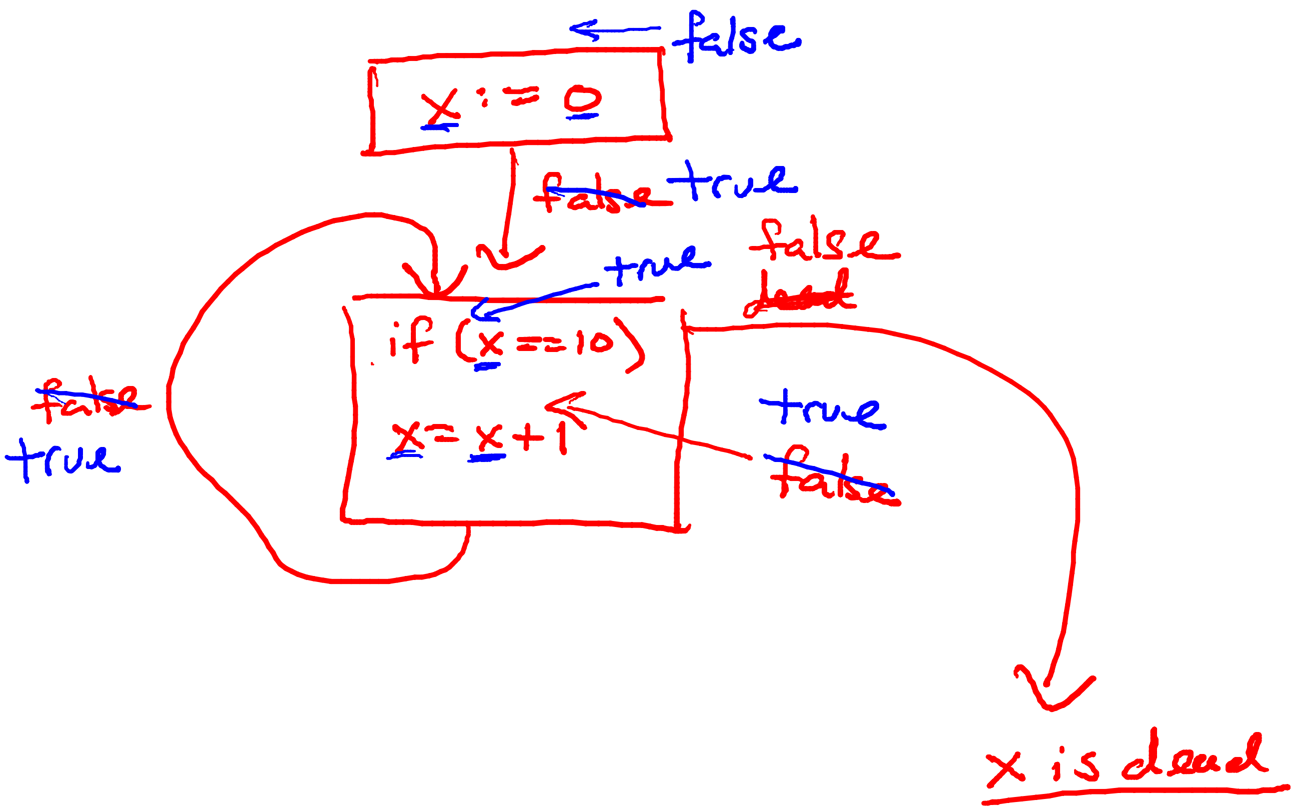


4. R4: `L(s, x, in) = L(s, x, out)` if `s` does not refer to `x`



1.6.2 Algorithm

1. Let all `L(...) = false` initially
2. Repeat until all statements s satisfy rules 1-4:
   - Pick `s` not satisfying 1-4 and update using the appropriate rule



Termination

- A value can change from false to true, but not the other way around
- Each value can change only once, so termination is guaranteed

1.6.3 Summary

2 kinds of analysis

1. Constant propagation is a **forwards analysis**: information is pushed from inputs to outputs
2. Liveness is a **backwards analysis**: information is pushed from outputs back towards inputs

Chapter 2

Memory Management / Garbage Collection

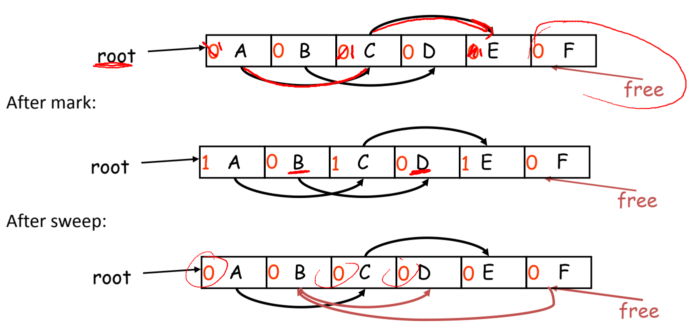
- new: allocate space
- Garbage Collection
- C and C++ programs have many memory-related bugs
  - double free , use after free, dangling pointer
  - overwrite part of data structure by accidents ...
  - OpenSSL Heartbleed
  - Apache Optionbleed
- Memory bugs are *REALY* hard to find
  - x = 3
  - y = 3
  - bugs happen in the FUTURE
- Automatic Memory Management
  - 1950s ...
  - Become mainstream with popularity of Java (1990's, Gosling?)
- Managing Memory
  - When an object is created, its runtime environment will allocate unused space for the object (new X)
  - after a while there will be no more unused space
  - Automatic MM attempt to determine which space is UNUSED (garbage) and automatically delete (free) it
  - How do we will when an object or space that object points to will never be used again ?
- Reachability Algorithms
  - A object X is reachable iff

* something (a variable) points to it
* another reachable object Y contains a pointer to X

– We can find all reachable objects by starting with all variables and follow their pointers
– An unreachable object can never be used, i.e., garbage

2.1 Mark and Sweep

- Mark and Sweep: when memory runs out, GC executes two phases:
  1. mark phase: traces all reachable objects
  2. sweep phase: collects garbage object
- Every object will have an extra bit: the *mark* bit
- Mark phase
  – initially all mark bit is 0
  – start from some root object (variable), traverse everything that variable can reach (point to)
    * mark those as 1
- Sweep phase
  – look at objects with mark bit 0 (garbage)
  – add them to a free list
  – objects with mark bit 1 reset to 0



- A serious problem with the mark phase
  – it is invoked when we are out of space
  – yet it needs space to construct the todo list
  – the size of the todo list is unbounded so we cannot reserve space for it a priori
- Solution:
  – The todo list is used as an auxiliary data structure to perform the reachability analysis
  – There is a trick that allows the auxiliary data to be stored in the objects themselves
    * **pointer reversal**: when a pointer is followed it is reversed to point to its parent
  – Similarly, the free list is stored in the free objects themselves

Pros and Cons

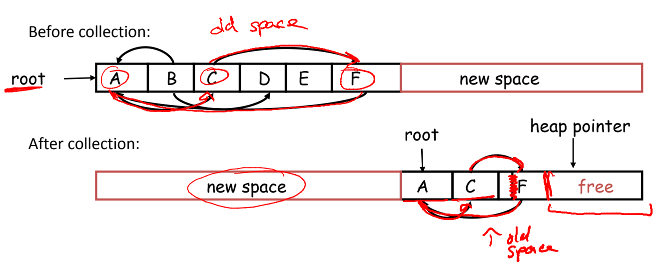
- Cons: Fragment memory
  - Space for a new object is allocated from the new list
  - a block large enough is picked
  - an area of the necessary size is allocated from it
  - the left-over is put back in the free list
- Pros: objects are not moved during GC
  - no need to update the pointers to objects
  - works for languages like C and C++

2.2 Stop and Copy

- Memory is organized into two areas
  - oldspace: used for allocation
  - new space: used as a reserve for GC
- The heap pointer points to the next free word in the old space
- Allocation just advances the heap pointer

Idea

- Starts when the old space is full
- Copies all reachable objects from old space into new space
  - garbage is left behind
  - after the copy phase the new space uses less space than the old one before the collection
- After the copy the roles of the old and new spaces are reversed and the program resumes



- We need to find all the reachable objects, as for mark and sweep
- As we find a reachable object we copy it into the new space
  - And we have to fix *ALL pointers* pointing to it!
- As we copy an object we store in the old copy a **forwarding pointer** to the new copy
  - when we later reach an object with a forwarding pointer we know it was already copied
  - same idea when we move to a new place, we place a fowarding address on the old address

Pros and Cons

- As with mark and sweep, we must be able to tell how large an object is when we scan it
  - and we must also know where the pointers are inside the object
- We must also copy any objects pointed to by the stack and update pointers in the stack
  - this can be an expensive operation
- Pros:
  - Stop and copy is generally believed to be the fastest GC technique
  - Allocation is very cheap (just increment the heap pointer)
  - Collection is relatively cheap
    - especially if there is a lot of garbage
    - only touch reachable objects
- Cons: some languages do not allow copying
  - C,C++

2.3 Reference Count

- Rather that wait for memory to be exhausted, try to collect an object when there are no more pointers to it
- **Reference Count**: Store in each object the number of pointers to that object
- Each assignment operation manipulates the reference count

Idea:

- `new` returns an object with reference count 1
- Let `rc(x)` be the reference count of `x`
- Assume `x, y` point to objects `o, p`
- Every assignment `x <- y` becomes:

```
rc(p) <- rc(p) + 1
rc(o) <- rc(o) - 1
if(rc(o) == 0) then free o
x <- y
```

Pros and Cons

- Pros:
  - easy to implement
  - collects garbage incrementally without large pauses in the execution
- Cons:
  - cannot collect circular structures (e.g., circularly linked list)
  - manipulating reference counts at each assignment is very slow

Chapter 3

Cool Extensions / Java

Additional notes

- https://www.cs.utexas.edu/~tdillig/cs345H/lecture18-6up.pdf

3.1 Java

- Java: COOL on steroids
- History of Java
  - Began as Oak at SUN
    - original target set-top devices
    - Initial development took several years ('91–'94)
  - Retargeted as the Internet language ('94–95)
    - Every new language needs a "killer app"
    - Alernatives such as TCL, Python
- Things that Cool does not have (and we will talk about how to extend Cool to add these features)
  - Arrays
  - Exceptions
  - Interfaces
  - Coercions
  - Dynamic Loading & Initialization
  - Threads
  - Summary
- Designs are based on
  - Modula-3 for types
  - Eiffel, ObjectiveC, C++ for Object orientation, interfaces
  - Lisp for Java's dynamic flavor (reflection)
- Java is a BIG language
  - Lots of features
  - Lots of feature *interactions*

3.2 (Java) Arrays

- Assume `B < A`. The following Java code

```
B[] b = new B[10];
A[] a = b;
a[0] = new A();
b[0].aMethodNotDeclaredInA();
```

  - pass the type checker
  - but gives runtime type error
  - Thus, java type system is unsound
  - Having multiple *aliases* to updateable locations with different types is *unsound!*

- Standard solution
  - Disallow subtyping through arrays

```
B < A if B inherits from A
C < A if C < B and B < A
B[] < A[] if B = A
```

- Java fixes the problem by checking each array assignment at runtime for type correctness
  - Is the type of the object being assigned compatible with the type of the array?
  - Cons: Adds overhead on array computations
  - Pros: But note that arrays of primitive types, which are more widely-used, are unaffected (because Primitive types are not classes)

3.3 Java Exceptions

- Deep in a section of code, you encounter an unexpected error
  - Out of memory
  - A list that is supposed to be sorted is not, etc.
- Add a new type (class) of exceptions
- Add new forms `try something catch(x) cleanup throw exception`