Lecture 23

CIS 4521/5521: COMPILERS

Announcements

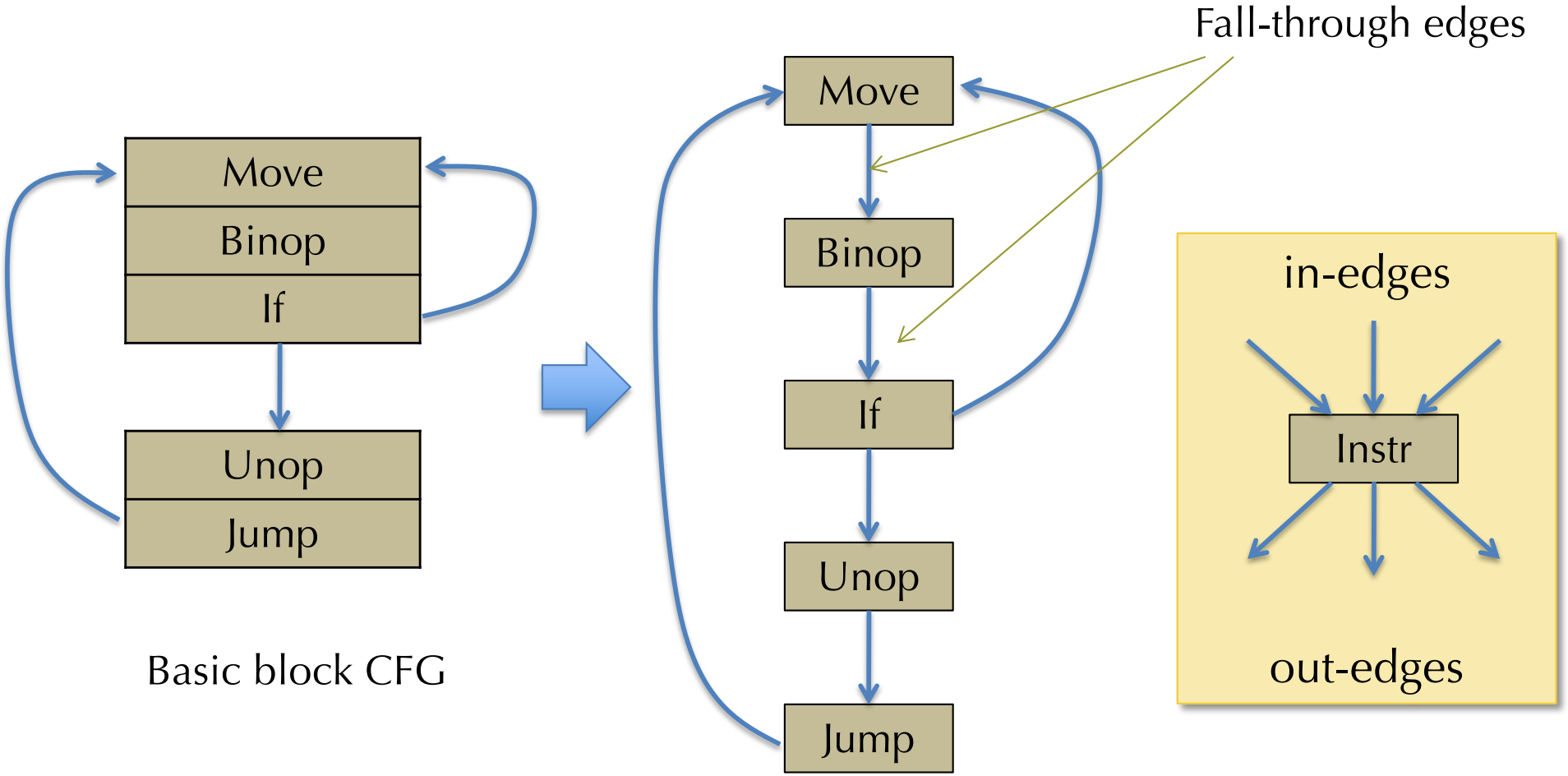
- HW6: Analysis & Optimizations
  - Alias analysis, constant propagation, dead code elimination, register allocation
  - Due: Wednesday, April 30th at 10:00pm
  - Posted test case due Tuesday, April 29th at 10:00pm

- Final Exam:
  - According to registrar: Thursday, May 8th noon - 2:00pm
  - Coverage: emphasizes material since the midterm
  - Cheat sheet: one, hand-written, double-sided, letter-sized page of notes

CODE ANALYSIS

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…



Example Liveness Analysis

Each iteration update:

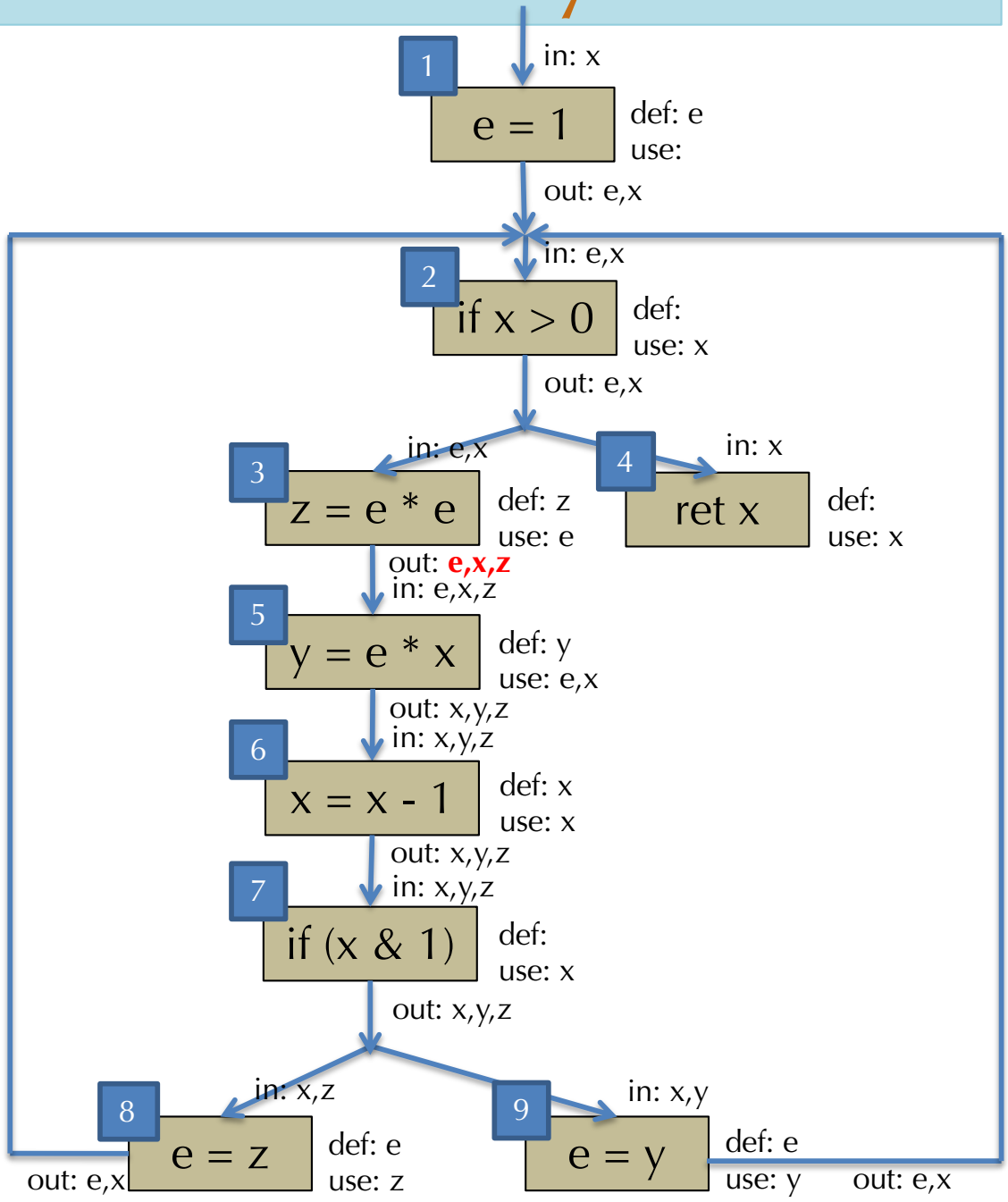
out[n] := $\cup_{n' \in succ[n]}$ in[n']

in[n] := use[n] ∪ (out[n] - def[n])

- Iteration 5:

out[3]= e,x,z

Done!



GENERAL DATAFLOW ANALYSIS

A Worklist Algorithm

- Use a FIFO queue of nodes that might need to be updated.

```
for all n, in[n] := ∅, out[n] := ∅
w = new queue with all nodes
repeat until w is empty
    let n = w.pop()                          // pull a node off the queue
     old_in = in[n]                          // remember old in[n]
     out[n] := ⋃_{n'∈succ[n]} in[n']
     in[n] := use[n] ∪ (out[n] - def[n])
     if (old_in != in[n]),                   // if in[n] has changed
        for all m in pred[n], w.push(m)      // add to worklist
end
```

Comparing Dataflow Analyses

Liveness:

Facts: {set of uids live at a program point }

let gen[n] = use[n] and kill[n] = def[n]

- $out[n] := \bigcup_{n' \in succ[n]} in[n']$ (backward)
- $in[n] := gen[n] \cup (out[n] - kill[n])$

Reaching Definitions:

Facts: {set of defns. that reach a program point}

let gen[n] = {n} and kill[n] = def[n]\{n}

- $in[n] := \bigcup_{n' \in pred[n]} out[n']$ (forward)
- $out[n] := gen[n] \cup (in[n] - kill[n])$

Available Expressions:

Facts: {set of rhs exps. that reach a program point}

e.g. gen[n] = {n}\kill[n] and kill[n] = use[n]

- $in[n] := \bigcap_{n' \in pred[n]} out[n']$ (forward)
- $out[n] := gen[n] \cup (in[n] - kill[n])$

Comparing Dataflow Analyses

Liveness:

Facts: {set of uids live at a program point }

let gen[n] = use[n] and kill[n] = def[n]

- out[n] := $\bigcup_{n' \in succ[n]}$ in[n′] (backward)
- in[n] := gen[n] ∪ (out[n] - kill[n])

Reaching Definitions:

Facts: {set of defns. that reach a program point}

let gen[n] = {n} and kill[n] = def[n]\{n}

- in[n] := $\bigcup_{n' \in pred[n]}$ out[n′] (forward)
- out[n] := gen[n] ∪ (in[n] - kill[n])

Available Expressions:

Facts: {set of rhs exps. that reach a program point}

e.g. gen[n] = {n}\kill[n] and kill[n] = use[n]

- in[n] := $\bigcap_{n' \in pred[n]}$ out[n′] (forward)
- out[n] := gen[n] ∪ (in[n] - kill[n])

Each analysis solves constraints over some ***domain*** of facts.

Comparing Dataflow Analyses

Liveness:

Facts: {set of uids live at a program point }

let gen[n] = use[n] and kill[n] = def[n]

- $out[n] := \bigcup_{n' \in succ[n]} in[n']$ (backward)
- $in[n] := gen[n] \cup (out[n] - kill[n])$

The "flow function" (i.e. effect of an instruction on the facts) can often be defined by gen and kill.

Reaching Definitions:

Facts: {set of defns. that reach a program point}

let gen[n] = {n} and kill[n] = def[n]\{n}

- $in[n] := \bigcup_{n' \in pred[n]} out[n']$ (forward)
- $out[n] := gen[n] \cup (in[n] - kill[n])$

Available Expressions:

Facts: {set of rhs exps. that reach a program point}

e.g. gen[n] = {n}\kill[n] and kill[n] = use[n]

- $in[n] := \bigcap_{n' \in pred[n]} out[n']$ (forward)
- $out[n] := gen[n] \cup (in[n] - kill[n])$

Comparing Dataflow Analyses

Liveness:

Facts: {set of uids live at a program point }

let gen[n] = use[n] and kill[n] = def[n]

- out[n] := $\bigcup_{n' \in succ[n]}$ in[n′] (backward)
- in[n] := gen[n] ∪ (out[n] - kill[n])

Backward analyses define out[] in terms of in[].

Reaching Definitions:

Facts: {set of defns. that reach a program point}

let gen[n] = {n} and kill[n] = def[n]\{n}

- in[n] := $\bigcup_{n' \in pred[n]}$ out[n′] (forward)
- out[n] := gen[n] ∪ (in[n] - kill[n])

Forward analyses define in[] in terms of out[].

Available Expressions:

Facts: {set of rhs exps. that reach a program point}

e.g. gen[n] = {n}\kill[n] and kill[n] = use[n]

- in[n] := $\bigcap_{n' \in pred[n]}$ out[n′] (forward)
- out[n] := gen[n] ∪ (in[n] - kill[n])

Comparing Dataflow Analyses

Liveness:

Facts: {set of uids live at a program point }

let gen[n] = use[n] and kill[n] = def[n]

- out[n] := $\bigcup_{n' \in succ[n]}$ in[n′] (backward)
- in[n] := gen[n] ∪ (out[n] - kill[n])

Reaching Definitions:

Facts: {set of defns. that reach a program point}

let gen[n] = {n} and kill[n] = def[n]\{n}

- in[n] := $\bigcup_{n' \in pred[n]}$ out[n′] (forward)
- out[n] := gen[n] ∪ (in[n] - kill[n])

Each domain of facts comes equipped with a way of aggregating information.

Available Expressions:

Facts: {set of rhs exps. that reach a program point}

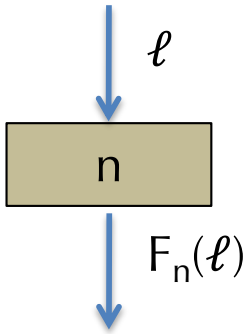
e.g. gen[n] = {n}\kill[n] and kill[n] = use[n]

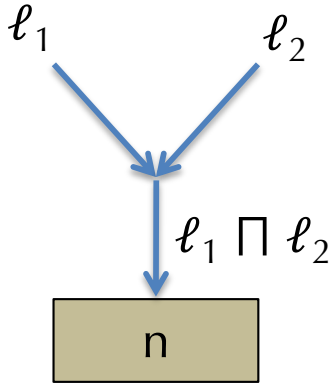
- in[n] := $\bigcap_{n' \in pred[n]}$ out[n′] (forward)
- out[n] := gen[n] ∪ (in[n] - kill[n])

(Forward) Dataflow Analysis Framework

A forward dataflow analysis can be characterized by:

1. A domain of dataflow values $\mathcal{L}$
   - e.g. $\mathcal{L}$ = the powerset of all variables
   - Think of $\ell \in \mathcal{L}$ as a property, then "x ∈ ℓ" means "x has the property"
2. For each node n, a flow function $F_n : \mathcal{L} \rightarrow \mathcal{L}$
   - So far we've seen $F_n(\ell) = gen[n] \cup (\ell - kill[n])$
   - So: $out[n] = F_n(in[n])$
   - "If ℓ is a property that holds before the node n, then $F_n(\ell)$ holds after n"
3. A combining operator $\prod$
   - "If we know *either* ℓ_1 *or* ℓ_2 holds on entry to node n, we know at most $\ell_1 \prod \ell_2$"
   - $in[n] := \prod_{n' \in pred[n]} out[n']$





Generic Iterative (Forward) Analysis

for all n, in[n] := $\top$, out[n] := $\top$
repeat until no change
    for all n

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

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

    end
end

- Here, $\top \in \mathcal{L}$ ("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 $\mathcal{L}$

- 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 $\ell_1 \supseteq \ell_2$ and we have $\top = \emptyset$

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 $\ell_1 \subseteq \ell_2$ and we have $\top = \{\text{set of all nodes}\}$

$\mathcal{L}$ as a Partial Order

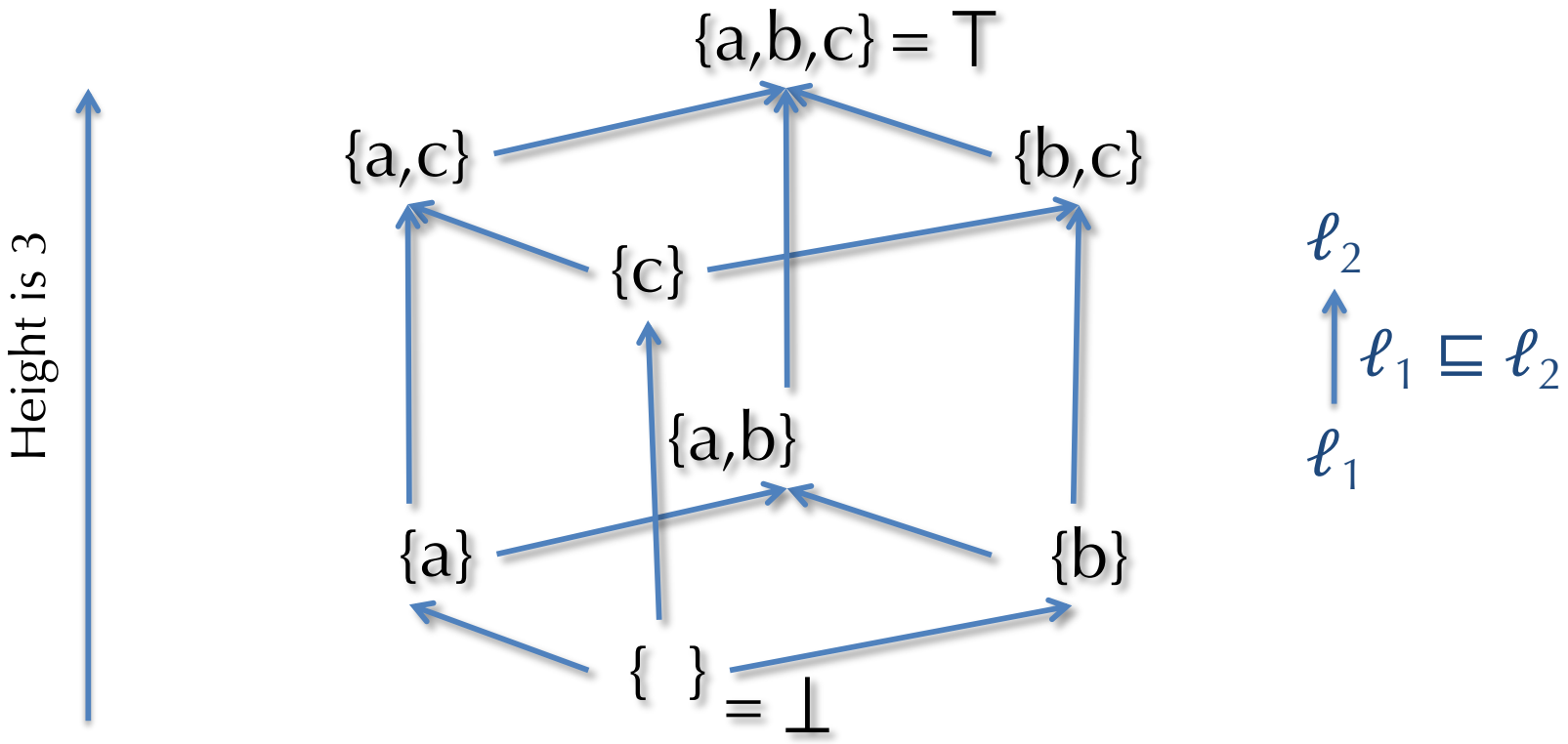
- $\mathcal{L}$ is a *partial order* defined by the ordering relation $\sqsubseteq$.
- A partial order is an ordered set.
- Some of the elements might be *incomparable*.
  - That is, there might be $\ell_1, \ell_2 \in \mathcal{L}$ such that neither $\ell_1 \sqsubseteq \ell_2$ nor $\ell_2 \sqsubseteq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

- 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 $\leq$
  - Types ordered by $<$:
  - Sets ordered by $\subseteq$ or $\supseteq$

Subsets of {a,b,c} ordered by ⊆

Partial order presented as a Hasse diagram.



order ⊑ is ⊆ meet ⊓ is ∩ join ⊔ is ∪

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"

Note: duality means that we can always "flip" a meet semi-lattice upside down to get a join semi-lattice (and vice-versa).

- 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*.
  - If it has just joins, it's a *join semi-lattice*.

Another Way to Describe the Algorithm

- Algorithm repeatedly computes (for each node n):
- $out[n] := F_n(in[n])$

- Equivalently: $out[n] := F_n(\bigsqcap_{n' \in pred[n]} out[n'])$
  - 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 $\mathbf{X} = (x_1, x_2, \ldots, x_n)$ it's a vector of points in $\mathcal{L}$
  - $\mathbf{F}(\mathbf{X}) = (F_1(\bigsqcap_{j \in pred[1]} out[j]), F_2(\bigsqcap_{j \in pred[2]} out[j]), \ldots, F_n(\bigsqcap_{j \in pred[n]} out[j]))$

- Any solution to the constraints is a *fixpoint* **X** of **F**
  - i.e. $\mathbf{F}(\mathbf{X}) = \mathbf{X}$

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*:
- $F : \mathcal{L} \rightarrow \mathcal{L}$ 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: $\mathbf{F} : \mathcal{L}^n \rightarrow \mathcal{L}^n$
  - 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

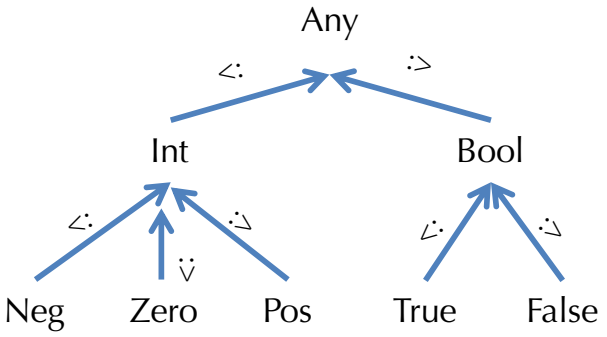
- Note that **F** is consistent: $\mathbf{F}(\mathbf{X}_0) \sqsubseteq \mathbf{X}_0$
  - So each iteration moves at least one step down the lattice (for some component of the vector)
  - $\ldots \sqsubseteq \mathbf{F}(\mathbf{F}(\mathbf{X}_0)) \sqsubseteq \mathbf{F}(\mathbf{X}_0) \sqsubseteq \mathbf{X}_0$

- Therefore, # steps needed to reach a fixpoint is at most the height H of $\mathcal{L}$ times the number of nodes: O(Hn)

Building Lattices?

- Information about individual nodes or variables can be lifted *pointwise:*
  - If $\mathcal{L}$ is a lattice, then so is $\{ f : X \rightarrow \mathcal{L} \}$ 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.

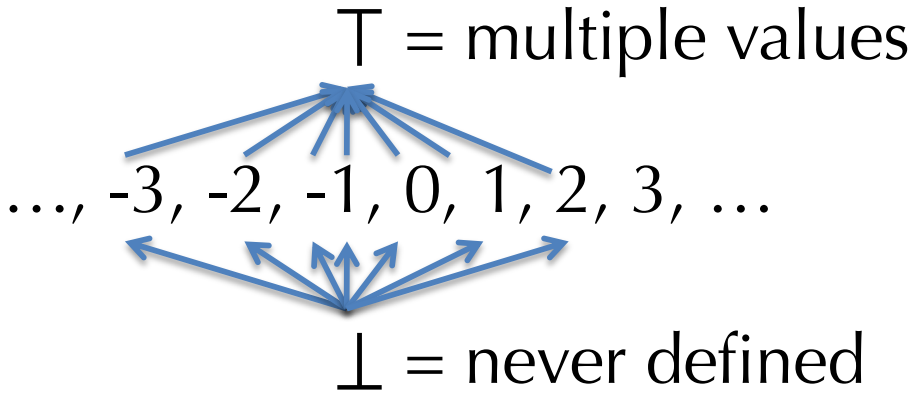
- Idea: propagate and fold integer constants in one pass:

```
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 $\mathcal{L}$ 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 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."

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

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

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

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

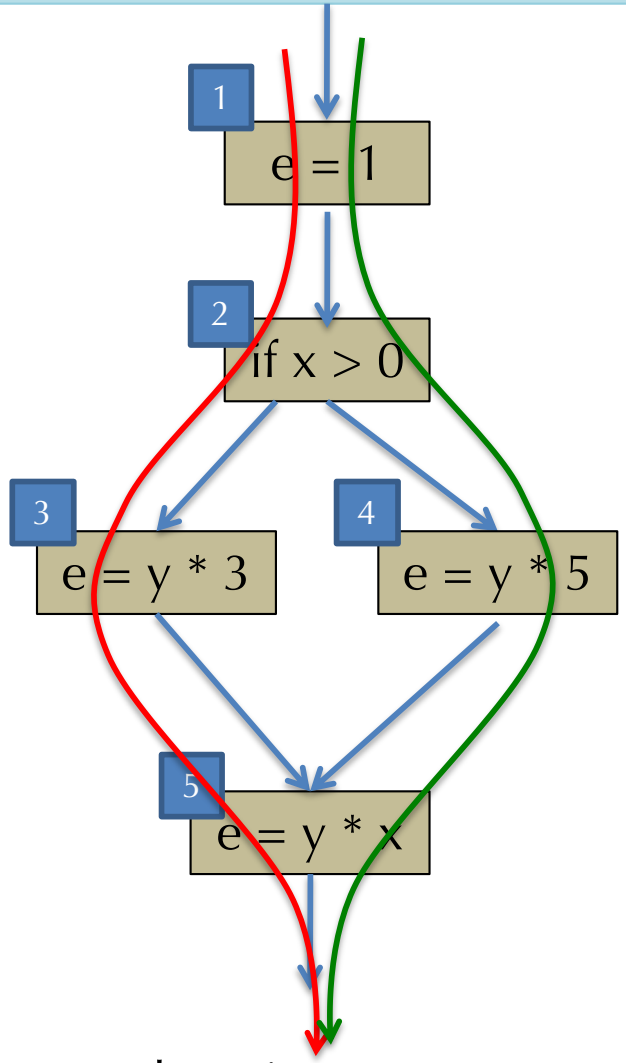
- Flow functions for the other nodes are easy…
- Monotonic?
- Distributes over meets?

QUALITY OF DATAFLOW ANALYSIS SOLUTIONS

Best Possible Solution

- Suppose we have a control-flow graph.
- If there is a path p_1 starting from the root node (entry point of the function) traversing the nodes
  $n_0, n_1, n_2, \ldots n_k$
- The best possible information along the path p_1 is:
  $\ell_{p1} = F_{nk}(\ldots F_{n2}(F_{n1}(F_{n0}(T)))\ldots)$
- Best solution at the output is some $\ell \sqsubseteq \ell_p$ for *all* paths p.

- Meet-over-paths (MOP) solution:
  $\prod_{p \in paths\_to[n]} \ell_p$



Best answer here is:

$F_5(F_3(F_2(F_1(T)))) \sqcap F_5(F_4(F_2(F_1(T))))$

What about quality of iterative solution?

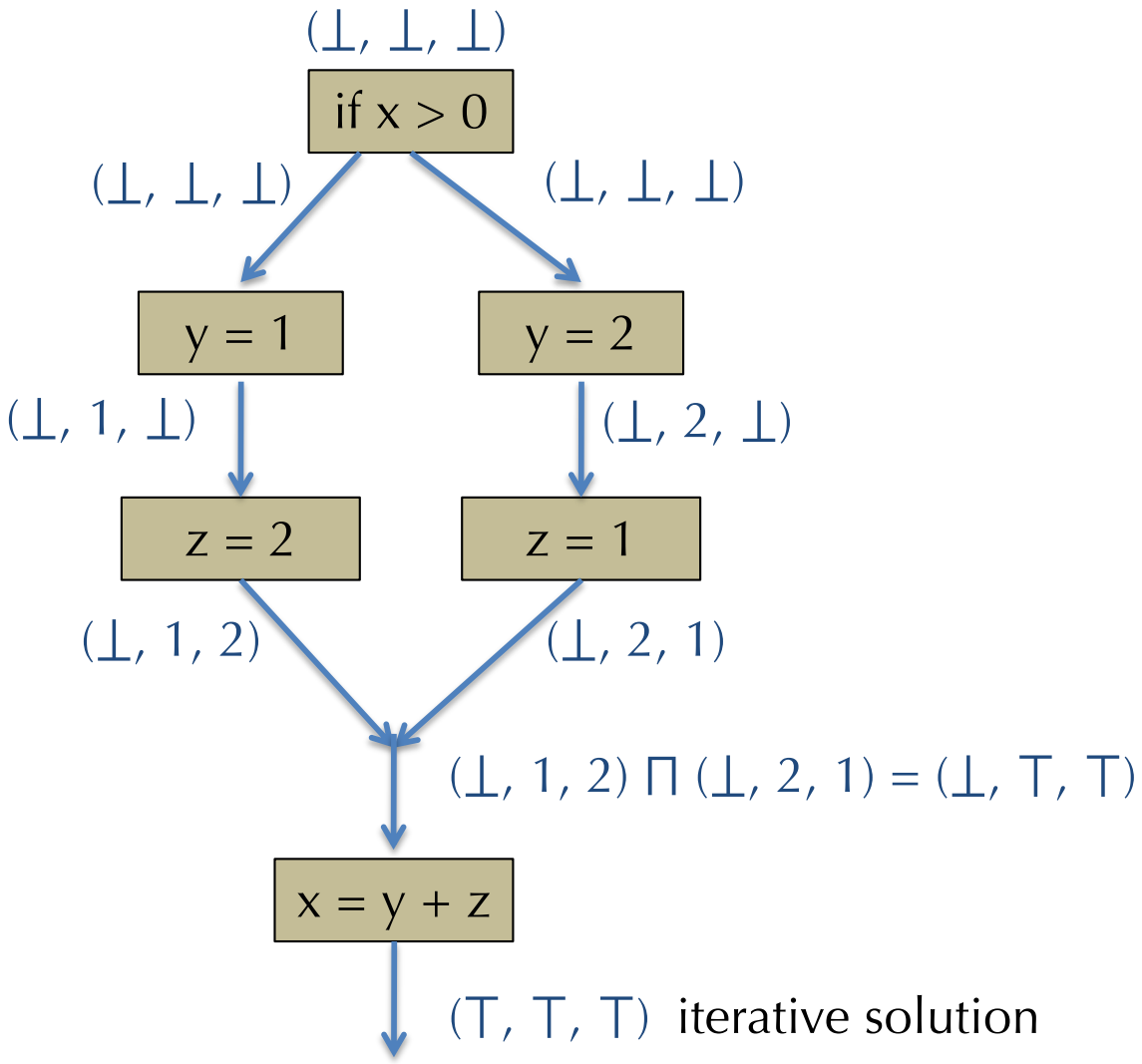
- Does the iterative solution: $out[n] = F_n(\bigsqcap_{n' \in pred[n]} out[n'])$ compute the MOP solution?
- MOP Solution: $\bigsqcap_{p \in paths\_to[n]} \ell_p$
- Answer: Yes, *if* the flow functions *distribute* over $\bigsqcap$
  - Distributive means: $\bigsqcap_i F_n(\ell_i) = F_n(\bigsqcap_i \ell_i)$
  - Proof is a bit tricky & beyond the scope of this class. (Difficulty: loops in the control flow graph might mean there are *infinitely* many paths…)
- Not all analyses give MOP solution
  - They are more conservative.

Reaching Definitions is MOP

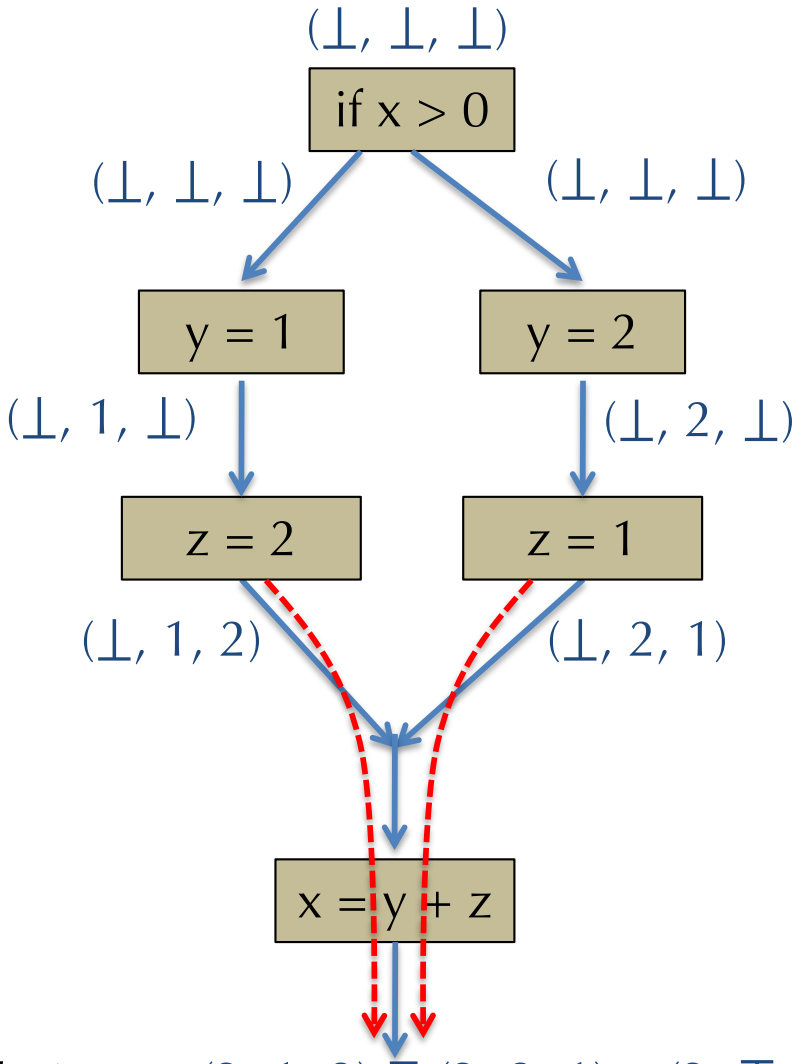
- $F_n[x] = gen[n] \cup (x - kill[n])$
- Does F_n distribute over meet $\sqcap$ $= \cup$?
- $F_n[x \sqcap y]$
  $= gen[n] \cup ((x \cup y) - kill[n])$
  $= gen[n] \cup ((x - kill[n]) \cup (y - kill[n]))$
  $= (gen[n] \cup (x - kill[n])) \cup (gen[n] \cup (y - kill[n]))$
  $= F_n[x] \cup F_n[y]$
  $= F_n[x] \sqcap F_n[y]$

- Therefore: Reaching Definitions with iterative analysis always terminates with the MOP (i.e. best) solution.

Constprop Iterative Solution



MOP Solution ≠ Iterative Solution

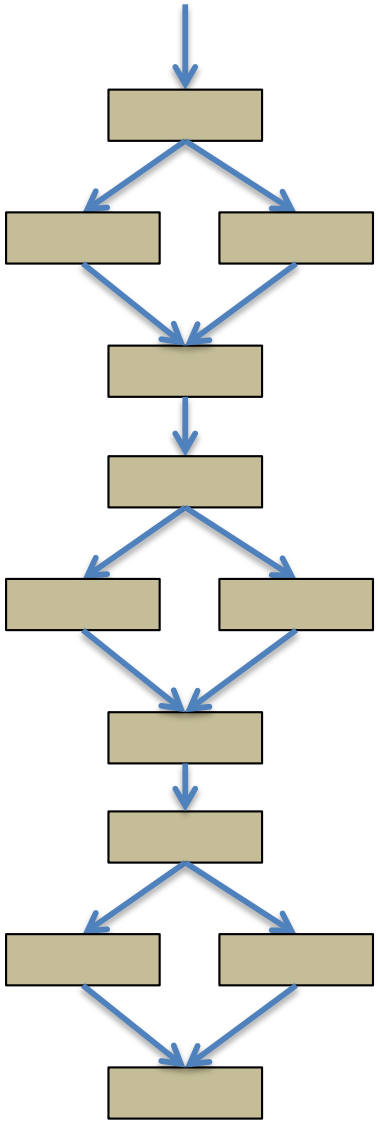


MOP solution $(3, 1, 2) \sqcap (3, 2, 1) = (3, \top, \top)$

Why not compute MOP Solution?

- If MOP is better than the iterative analysis, why not compute it instead?
  - ANS: exponentially many paths (even in graph without loops)

- O(n) nodes
- O(n) edges
- $O(2^n)$ paths*
  - At each branch there is a choice of 2 directions



* Incidentally, a similar idea can be used to force ML / Haskell type inference to need to construct a type that is exponentially big in the size of the program!

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.
  - Solution is equivalent to meet-over-paths answer if the flow functions distribute over meet ($\sqcap$).

- Dataflow analyses as presented work for an "imperative" intermediate representation.
  - The values of temporary variables are updated ("mutated") during evaluation.
  - Such mutation complicates calculations
  - SSA = "Single Static Assignment" eliminates this problem, by introducing more temporaries – each one assigned to only once.
  - Next up: Converting to SSA, finding loops and dominators in CFGs

See HW6: Dataflow Analysis

IMPLEMENTATION

REGISTER ALLOCATION

Register Allocation Problem

- Given: an IR program that uses an unbounded number of temporaries
  - e.g. the uids of our LLVM programs

- Find: a mapping from temporaries to machine registers such that
  - program semantics is preserved (i.e. the behavior is the same)
  - register usage is maximized
  - moves between registers are minimized
  - calling conventions / architecture requirements are obeyed

- Stack Spilling
  - If there are k registers available and m > k temporaries are live at the same time, then not all of them will fit into registers.
  - So: "spill" the excess temporaries to the stack.

Linear-Scan Register Allocation

Simple, greedy register-allocation strategy:

1. Compute liveness information: `live(x)`
   - recall: `live(x)` is the set of uids that are live on entry to `x`'s definition
2. Let `pal` be the set of usable registers
   - usually reserve a couple for spill code [our implementation uses rax,rcx]
3. Maintain "layout" `uid_loc` that maps uids to locations
   - locations include registers and stack slots n, starting at n=0
4. Scan through the program. For each instruction that defines a uid `x`
   - `used` = `{r | reg r = uid_loc(y)` s.t. `y ∈ live(x)}`
   - `available = pal - used`
   - If `available` is empty: *// no registers available, spill*
     `uid_loc(x) := slot n ; n = n + 1`
   - Otherwise, pick `r` in `available`: *// choose an available register*
     `uid_loc(x) := reg r`

For HW6

- HW 6 implements two naive register allocation strategies:
  - `none`: spill all registers
  - `greedy`: uses linear scan
- Also offers choice of liveness
  - `trivial`: assume all variables are live everywhere
  - `dataflow`: use the dataflow algorithms

- Your job: do "better" than these.
- Quality Metric:
  - first, minimize memory accesses
  - then prioritize for shorter code

- Linear scan is OK
  - but… how can we do better?

GRAPH COLORING

Register Allocation

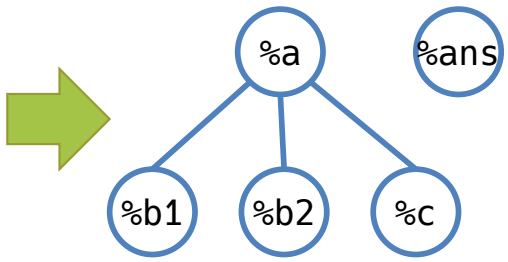
Basic process:

1. Compute liveness information for each temporary.
2. Create an *interference graph*:
   - Nodes are temporary variables.
   - There is an edge between node n and m if n is live at the same time as m
3. Try to color the graph
   - Each color corresponds to a register
4. In case step 3. fails, "spill" a register to the stack and repeat the whole process.
5. Rewrite the program to use registers

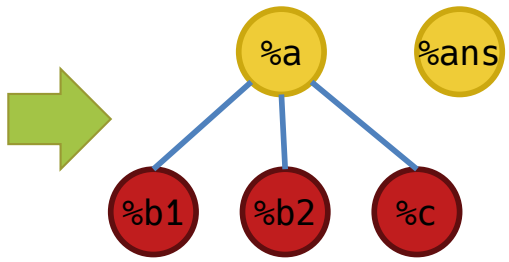
Interference Graphs

- Nodes of the graph are `%uids`
- Edges connect variables that *interfere* with each other
  - Two variables interfere if their live ranges intersect (i.e. there is an edge in the control-flow graph across which they are both live).
- Register assignment is a *graph coloring*.
  - A graph coloring assigns each node in the graph a color (register)
  - Any two nodes connected by an edge must have different colors.
- Example:

```
// live = {%a}
%b1 = add i32 %a, 2
// live = {%a,%b1}
%c = mult i32 %b1, %b1
// live = {%a,%c}
%b2 = add i32 %c, 1
// live = {%a,%b2}
%ans = mult i32 %b2, %a
// live = {%ans}
return %ans;
```



Interference Graph



2-Coloring of the graph
red = r8
yellow = r9

Register Allocation Questions

- Can we efficiently find a k-coloring of the graph whenever possible?
  - Answer: in general the problem is NP-complete (it requires search)
  - But, we can do an efficient approximation using heuristics.

- How do we assign registers to colors?
  - If we do this in a smart way, we can eliminate redundant MOV instructions.

- What do we do when there aren't enough colors/registers?
  - We have to use stack space, but how do we do this effectively?

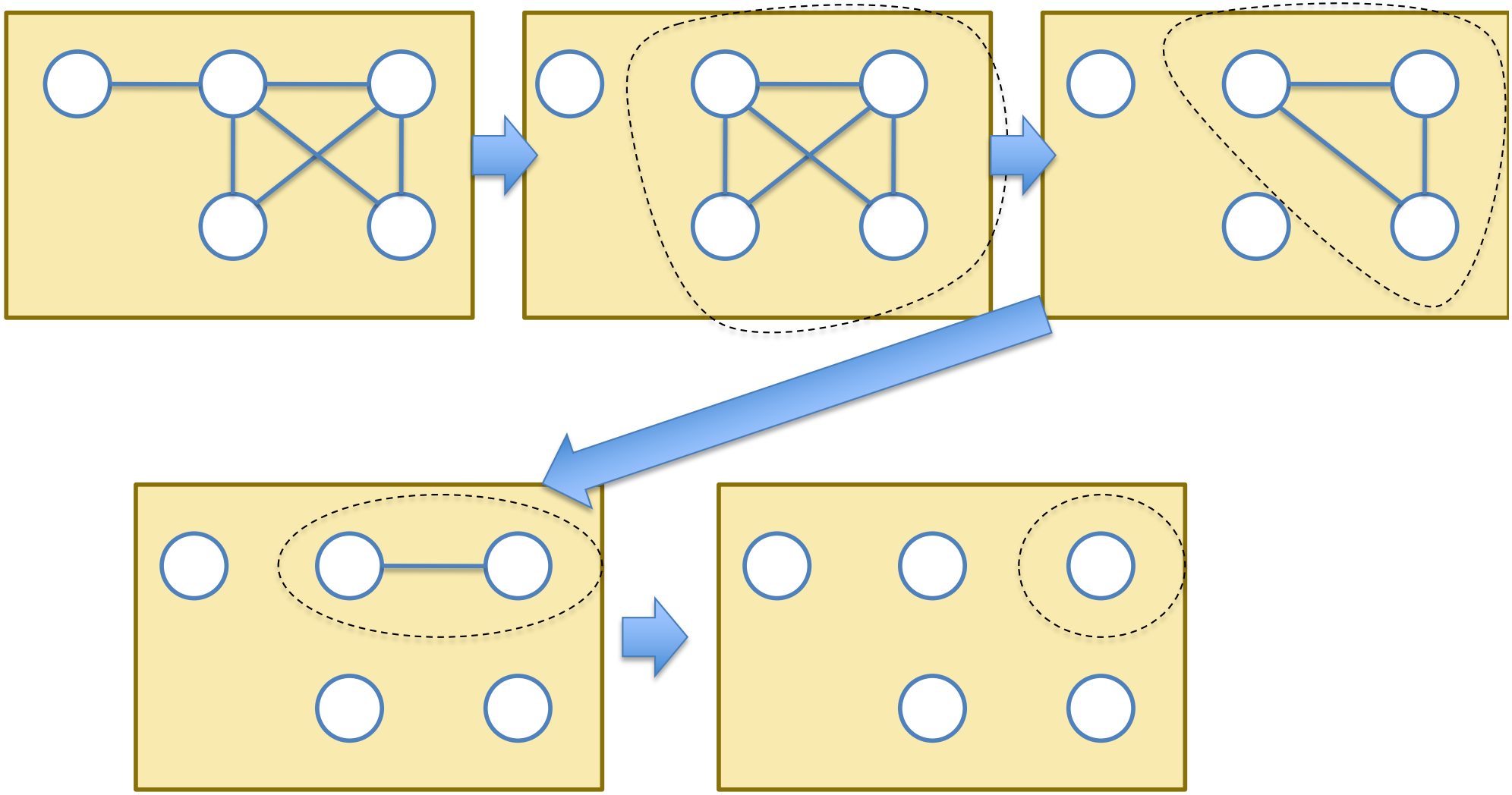
Coloring a Graph: Kempe's Algorithm

Kempe [1879] provides this algorithm for K-coloring a graph.

It's a recursive algorithm that works in three steps:

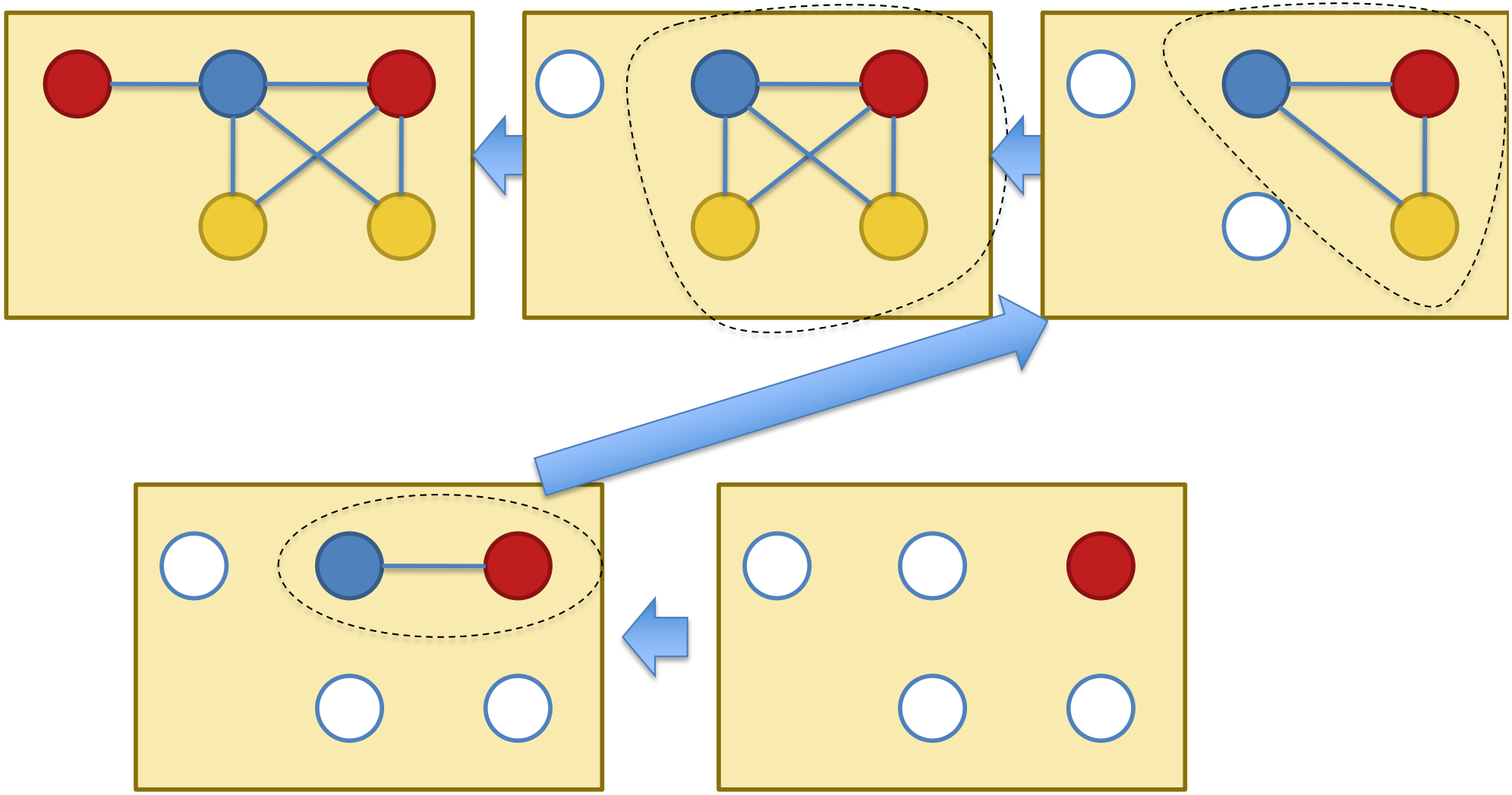
1. Find a node with degree < K and cut it out of the graph.
   - Remove the nodes and edges.
   - This is called *simplifying* the graph
2. Recursively K-color the remaining subgraph
3. When remaining graph is colored, there must be at least one free color available for the deleted node (since its degree was < K). Pick such a color.

Example: 3-color this Graph



Recursing Down the Simplified Graphs

Example: 3-color this Graph

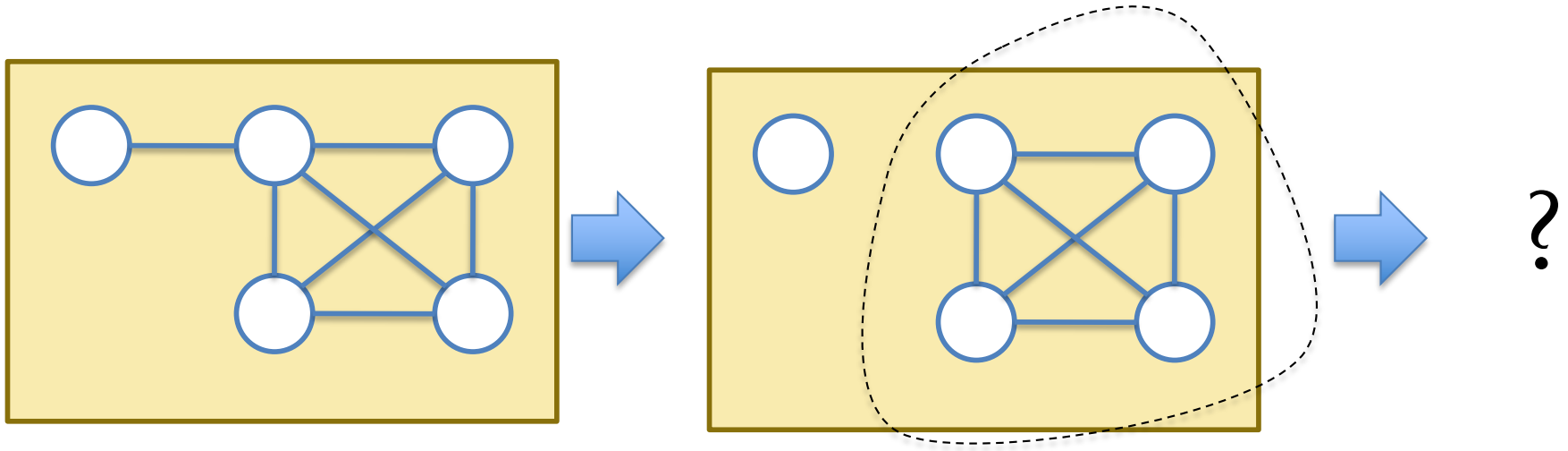


Assigning Colors on the way back up.

Failure of the Algorithm

- If the graph cannot be colored, it will simplify to a graph where every node has at least K neighbors.
  - This can happen even when the graph is K-colorable!
  - This is a symptom of NP-hardness (it requires search)

- Example: When trying to 3-color this graph:



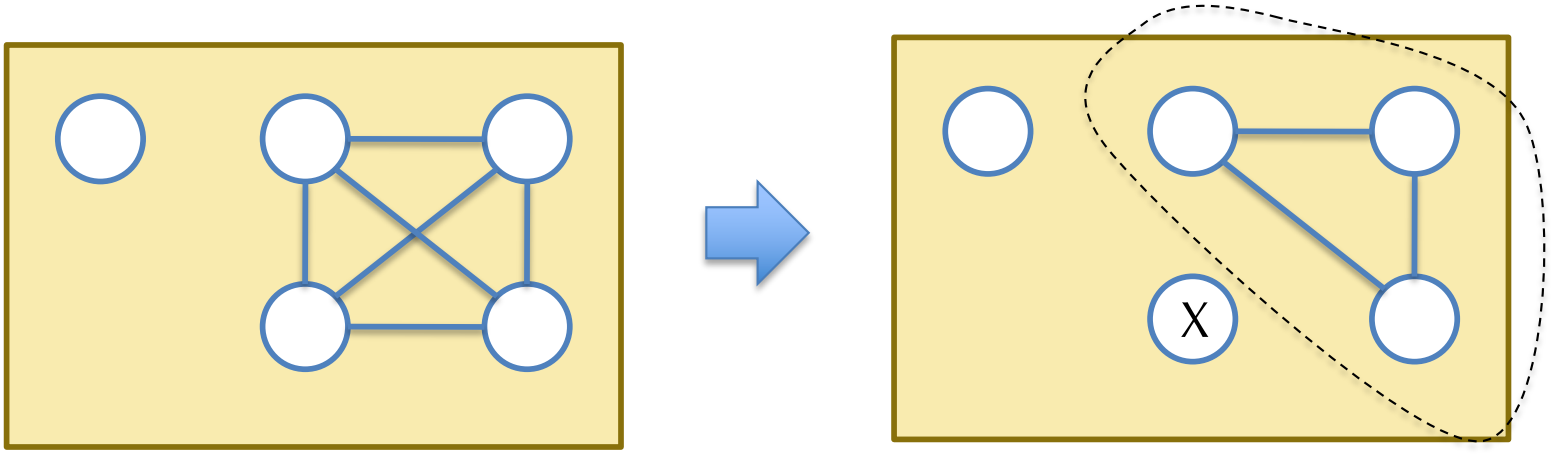
Spilling

- Idea: If we can't K-color the graph, we need to store one temporary variable on the stack.
- Which variable to spill?
  - Pick one that isn't used very frequently
  - Pick one that isn't used in a (deeply nested) loop
  - Pick one that has high interference
    (since removing it will make the graph easier to color)
- In practice: some weighted combination of these criteria

- When coloring:
  - Mark the node as spilled
  - Remove it from the graph
  - Keep recursively coloring

Spilling, Pictorially

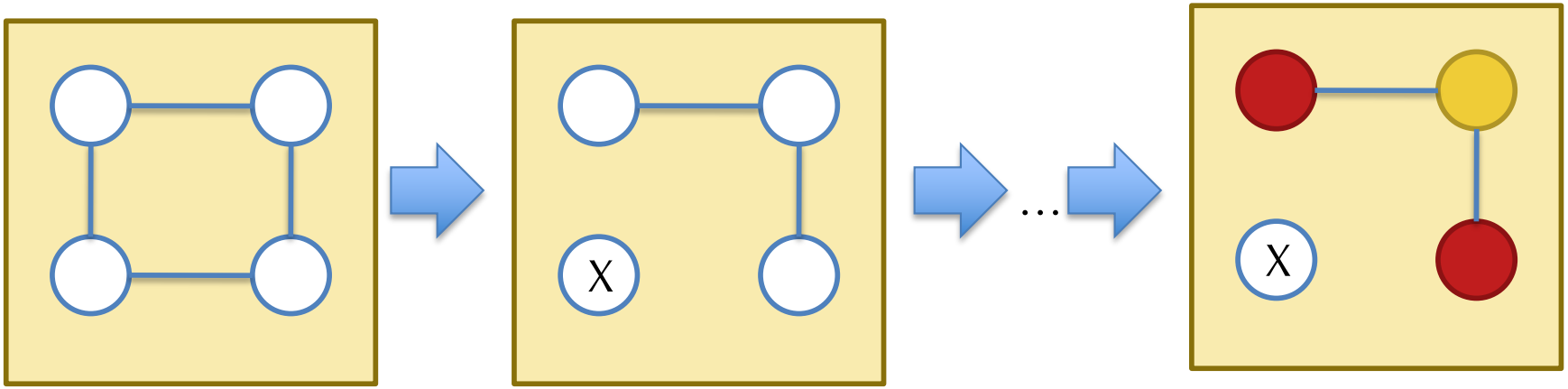
- Select a node to spill
- Mark it and remove it from the graph
- Continue coloring



Optimistic Coloring

- Sometimes it is possible to color a node marked for spilling.
  - If we get "lucky" with the choices of colors made earlier.

- Example: When 2-coloring this graph:



- Even though the node was marked for spilling, we can color it.
- So: on the way down, mark for spilling, but don't actually spill…

Accessing Spilled Registers

- If optimistic coloring fails, we need to generate code to move the spilled temporary to & from memory.
- Option 1: Reserve registers specifically for moving to/from memory.
  - Con: Need at least two registers (one for each source operand of an instruction), so decreases total # of available registers by 2.
  - Pro: Only need to color the graph once.
  - Not good on 32bit x86 because there are too few registers & too many constraints on how they can be used.
  - OK on 64bit x86 and other processors. (We use this for HW6)

- Option 2: Rewrite the program to use a new temporary variable, with explicit moves to/from memory.
  - Pro: Need to reserve fewer registers.
  - Con: Introducing temporaries changes live ranges, so must recompute liveness & recolor graph

Example Spill Code

- Suppose temporary `t` is marked for spilling to stack slot located at `[rbp+offs]`
- Rewrite the program like this:

```
t = a op b;                t = a op b                    // defn. of t
…                          Mov [rbp+offs], t
                           …
x = t op c                 Mov t37, [rbp+offs] // use 1 of t
…                          x = t37 op c
                           …
y = d op t                 Mov t38, [rbp+offs] // use 2 of t
                           y = d op t38
```

- Here, `t37` and `t38` are freshly generated temporaries that replace `t` for different uses of `t`.
- Rewriting the code in this way breaks `t`'s live range up:

  `t`, `t37`, `t38` are only live across one edge

Precolored Nodes

- Some variables must be pre-assigned to registers.
  - E.g. on X86 the multiplication instruction: IMul must define %rax
  - The "Call" instruction should kill the caller-save registers %rax, %rcx, %rdx.
  - Any temporary variable live across a call interferes with the caller-save registers.

- To properly allocate temporaries, we treat registers as nodes in the interference graph with pre-assigned colors.
  - Pre-colored nodes can't be removed during simplification.
  - Trick: Treat pre-colored nodes as having "infinite" degree in the interference graph – this guarantees they won't be simplified.
  - When the graph is empty except the pre-colored nodes, we have reached the point where we start coloring the rest of the nodes.

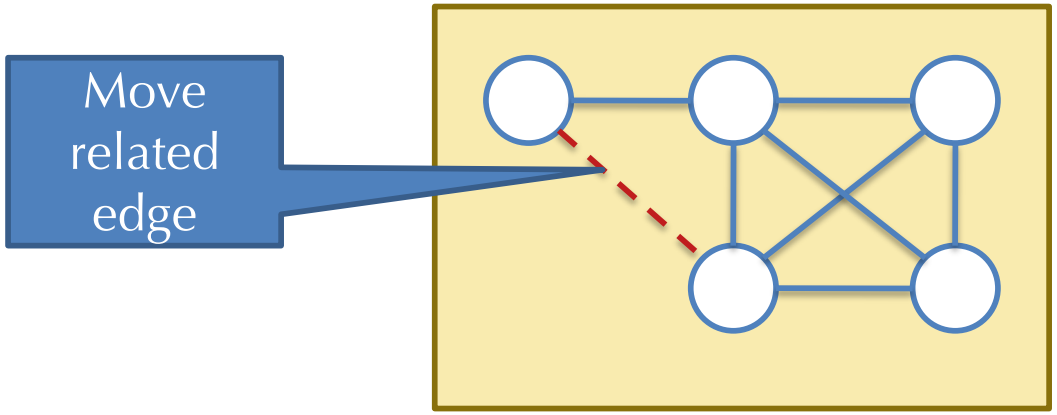
Picking Good Colors

- When choosing colors during the coloring phase, *any* choice is semantically correct, but some choices are better for performance.
- Example:
  `movq t1, t2`
  - If t1 and t2 can be assigned the same register (color) then this move is redundant and can be eliminated.

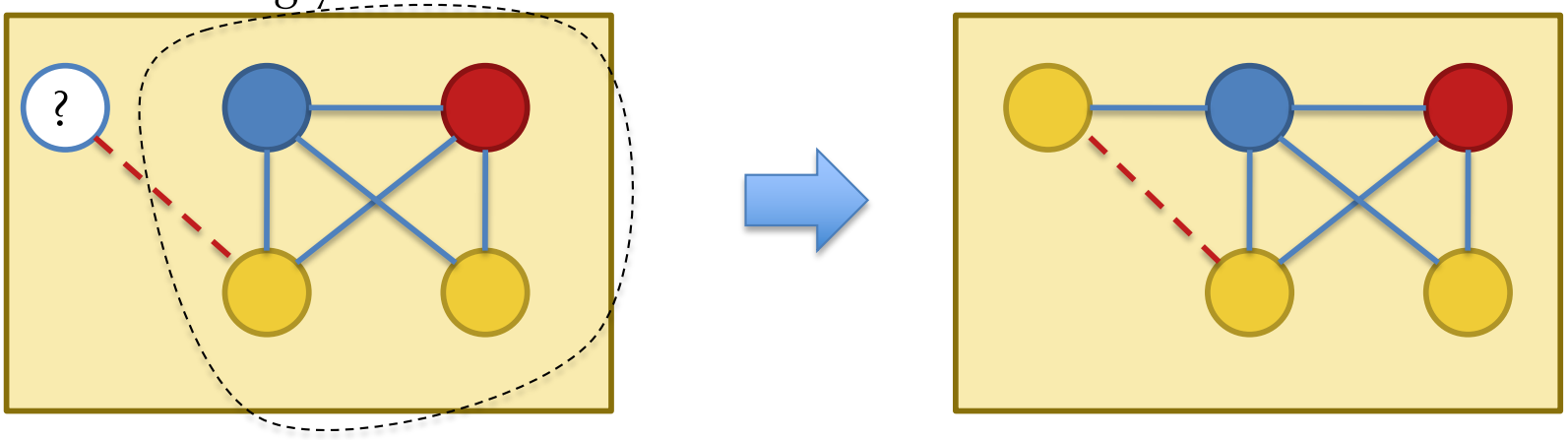
- A simple color choosing strategy that helps eliminate such moves:
  - Add a new kind of "move related" edge between the nodes for t1 and t2 in the interference graph.
  - When choosing a color for t1 (or t2), if possible, pick a color of an already colored node reachable by a move-related edge.

Example Color Choice

- Consider 3-coloring this graph, where the dashed edge indicates that there is a Mov from one temporary to another.

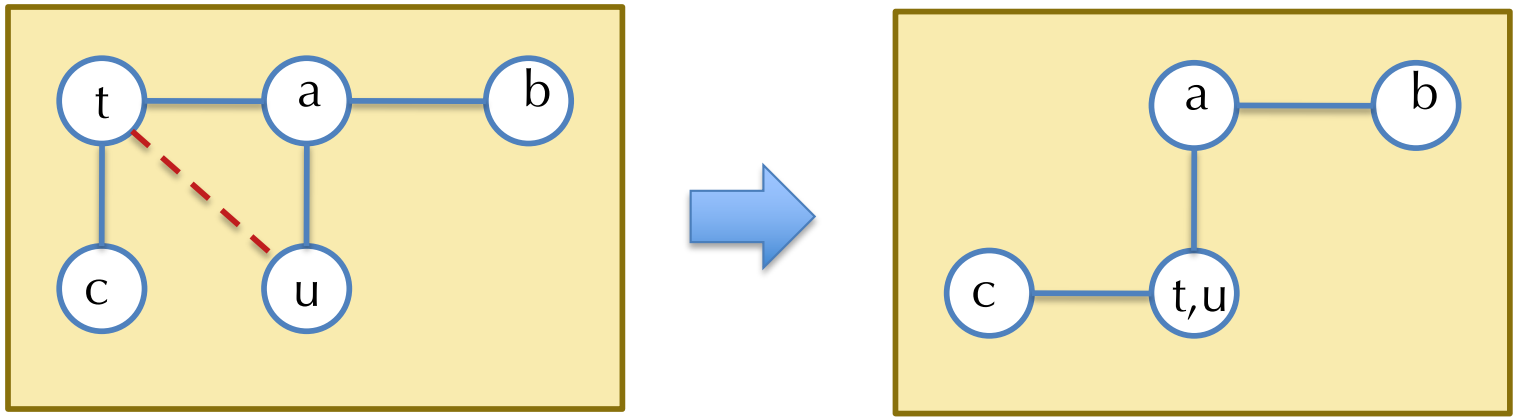


- After coloring the rest, we have a choice:
  - Picking yellow is better than red because it will eliminate a move.

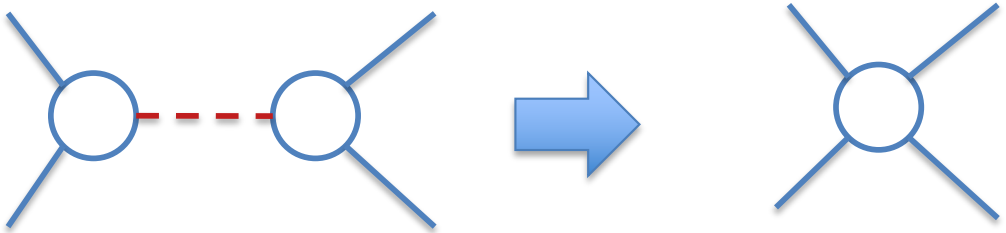


Coalescing Interference Graphs

- A more aggressive strategy is to *coalesce* nodes of the interference graph if they are connected by move-related edges.
  - Coalescing the nodes *forces* the two temporaries to be assigned the same register.



- Idea: interleave simplification and coalescing to maximize the number of moves that can be eliminated.
- Problem: coalescing can sometimes increase the degree of a node.



Conservative Coalescing

- Two strategies are guaranteed to preserve the k-colorability of the interference graph.

1. *Brigg's strategy*: It's safe to coalesce x & y if the resulting node will have fewer than k neighbors (with degree ≥ k).

2. *George's strategy:* We can safely coalesce x & y if for every neighbor t of x, either t already interferes with y or t has degree < k.

Complete Register Allocation Algorithm

1. Build interference graph (precolor nodes as necessary).
   - Add move related edges
2. Reduce the graph (building a stack of nodes to color).
   1. Simplify the graph as much as possible without removing nodes that are move related (i.e. have a move-related neighbor). Remaining nodes are high degree or move-related.
   2. Coalesce move-related nodes using Brigg's or George's strategy.
   3. Coalescing can reveal more nodes that can be simplified, so repeat 2.1 and 2.2 until no node can be simplified or coalesced.
   4. If no nodes can be coalesced *freeze* (remove) a move-related edge and keep trying to simplify/coalesce.
3. If there are non-precolored nodes left, mark one for spilling, remove it from the graph and continue doing step 2.
4. When only pre-colored node remain, start coloring (popping simplified nodes off the top of the stack).
   1. If a node must be spilled, insert spill code as on slide 14 and rerun the whole register allocation algorithm starting at step 1.

Last details

- After register allocation, the compiler should do a peephole optimization pass to remove redundant moves.
- Some architectures specify calling conventions that use registers to pass function arguments.
  - It's helpful to move such arguments into temporaries in the function prelude so that the compiler has as much freedom as possible during register allocation. (Not an issue on X86, though.)