Lecture 18:

Constraints 2: Implementations (with help from Scott Hudson)

05-431/631 Software Structures for User Interfaces (SSUI)

Fall, 2021

Logistics

- Homework 5 due on Thursday

Form of constraints

- For UI work, typically express in form of equations
  - Often just data-copying (equality): this.x = that.x
  - For graphics, usually arithmetic required:
    - this.x = that.x + that.w + 5
      - 5 pixels to the right
    - this.x = that.x + that.w/2 - this.w/2
      - centered
    - this.w = 10 + max child[i].x + child[i].w
      - 10 larger than children

Power of constraints

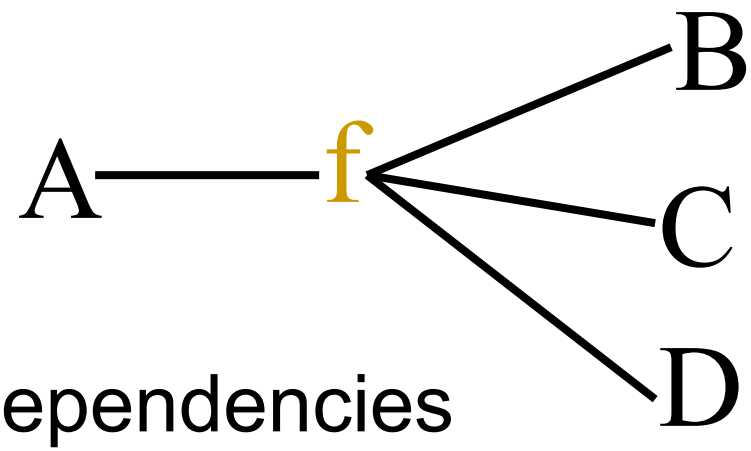
- If something changes, system can determine effects
  - automatically
  - just change the object that has to change, the rest "just happens"
    - very nice property

Dependency graphs

- Useful to look at a system of constraints as a "dependency graph"
  - graph showing what depends on what
  - two kinds of nodes (bipartite graph)
    - variables (values to be constrained)
    - constraints (equations that relate)

Dependency graphs

Example: A = f(B, C, D)

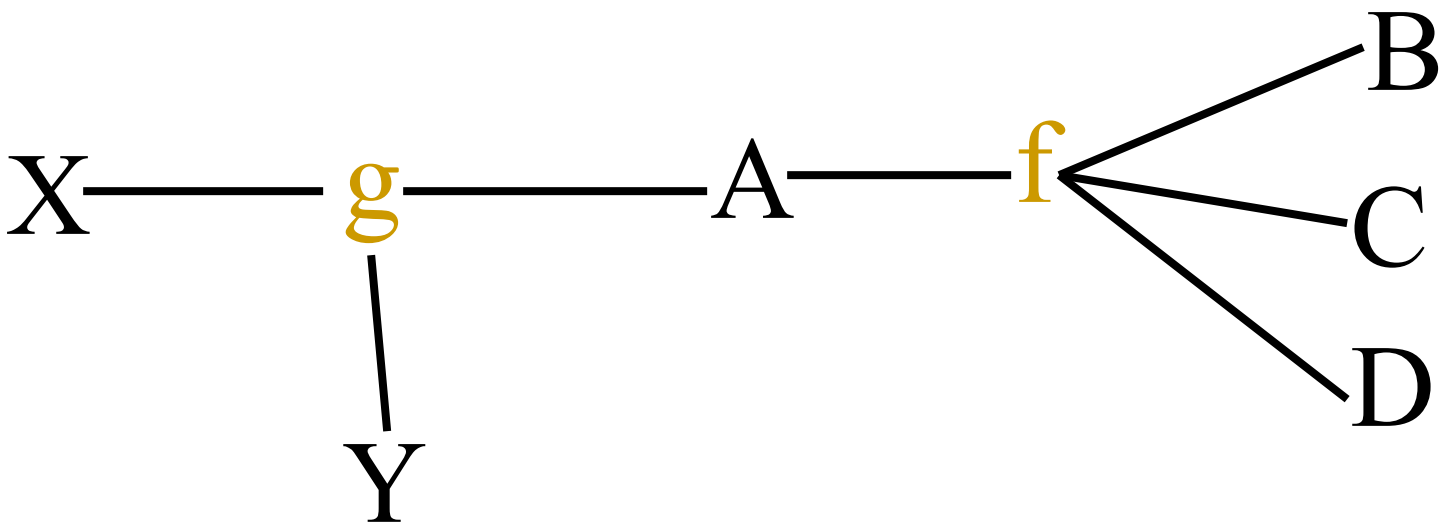


Edges are dependencies

Dependency graphs

Dependency graphs chain together:

X = g(A, Y)

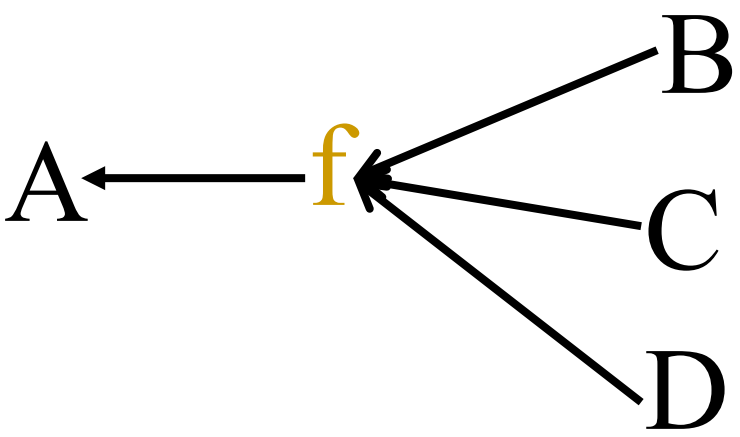


Kinds of constraint systems

- Actually lots of kinds, but 3 major varieties used in UI work
  - one-way, multi-way, numerical (less use)
  - reflect kinds of limitations imposed
  - Reminder: Angular has *both one-way* and *multi-way*
- One-Way constraints
  - must have a single variable on LHS
  - information only flows to that variable
    - can change B,C,D system will find A
    - can't do reverse (change A …)

One-Way constraints

Results in a directed dependency graph:
A = f(B,C,D)



NOTE: These arrows are in the ***dataflow*** direction. Not dependency

Normally require dependency graph to be acyclic

- cyclic graph means cyclic definition

One-Way constraints

- Problem with one-way: introduces an asymmetry
  - this.x = that.x + that.w + 5
  - can move “that” (change that.x) but can’t move “this”

Multi-way constraints

$A = f(B,C,D)$

Don't require info flow only to the left in equation

- can change A and have system find B,C, and/or D

Not as hard as it might seem

- most systems require you to explicitly factor the equations for them
  - provide B = g(A,C,D), etc.
- I believe this is true for Angular two-way bindings – have to supply a function for each "way" unless equality

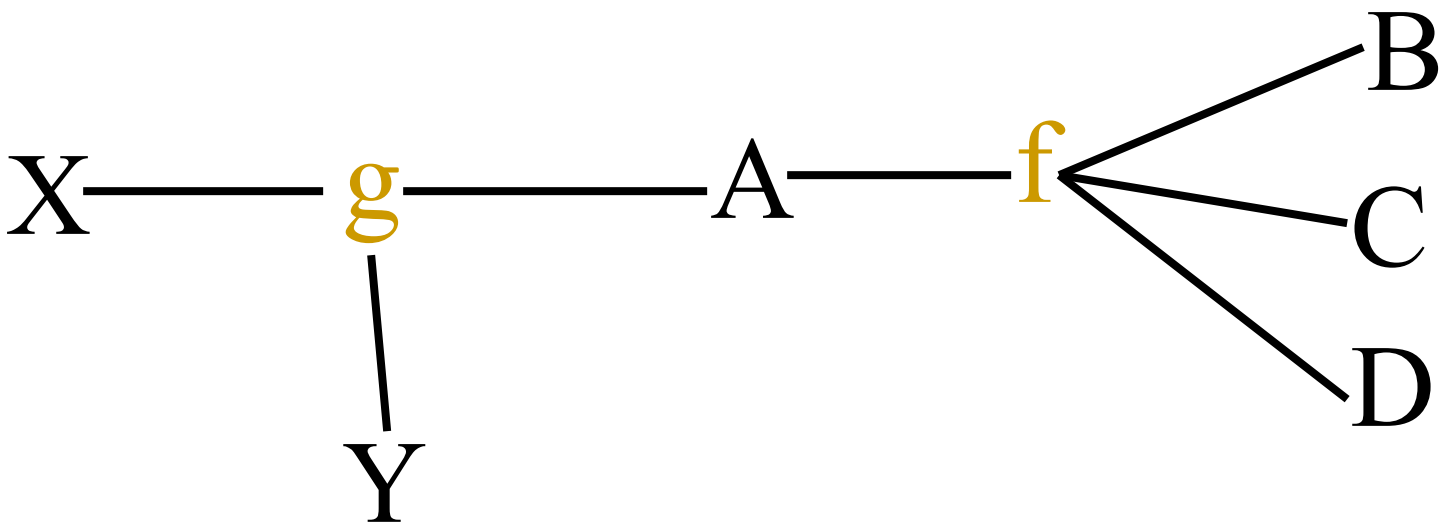
Multi-way constraints

- Modeled as an undirected dependency graph
- No longer have asymmetry

Multi-way constraints

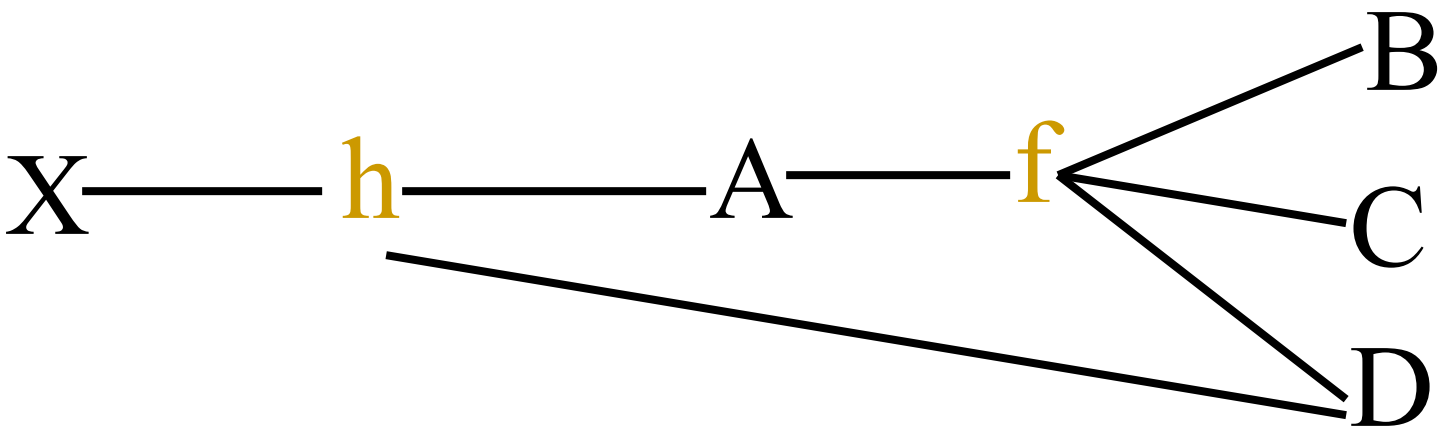
But all is not rosy

- most efficient algorithms require that dependency graph be a tree (acyclic undirected graph)



Multi-way constraints

But: A = f(B,C,D) & X = h(D,A)



Not OK because it has a cycle (not a tree)

Another important issue

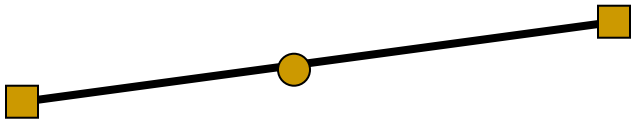
- A set of constraints can be:
  - Over-constrained
    - No valid solution that meets all constraints
  - Under-constrained
    - More than one solution
      - sometimes infinite numbers

Over- and under-constrained

- Over-constrained systems
  - solver will fail
  - isn't nice to do this in interactive systems
  - typically need to avoid this
    - need at least a "fallback" solution

Over- and under-constrained

- Under-constrained
  - many solutions
  - system has to pick one
  - may not be the one you expect
  - example: constraint: point stays at midpoint of line segment
    - move end point, then?



Over- and under-constrained

- Under-constrained
  - example: constraint: point stays at midpoint of line segment
    - move end point, then?
    - Lots of valid solutions
      - move other end point
      - collapse to one point
      - etc.

Over- and under-constrained

- Good news is that one-way is never over- or under-constrained (assuming acyclic)
  - system makes no arbitrary choices
  - pretty easy to understand

Over- and under-constrained

- Multi-way can be either over- or under-constrained
  - have to pay for extra power somewhere
  - typical approach is to over-constrain, but have a mechanism for breaking / loosening constraints in priority order
    - one way: “constraint hierarchies”

Over- and under-constrained

- Multi-way can be either over- or under-constrained
  - unfortunately system still has to make arbitrary choices
  - generally harder to understand and control

Implementing constraints

- Algorithm for one-way systems
  - Need bookkeeping for variables
  - For each keep:

    value - the value of the var

    eqn - code to eval constraint

    dep - list of vars we depend on

    ---

    done - boolean “mark” for alg

Implementing constraints

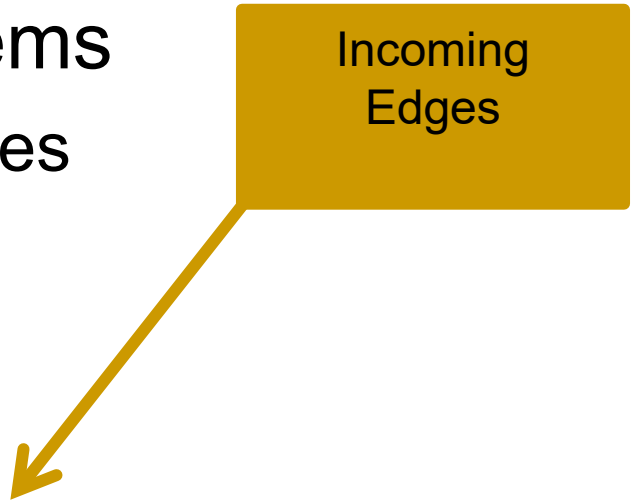
- Algorithm for one-way systems
  - Need bookkeeping for variables
  - For each keep:

    value - the value of the var

    eqn - code to eval constraint

    dep - list of vars we depend on

    done - boolean “mark” for alg



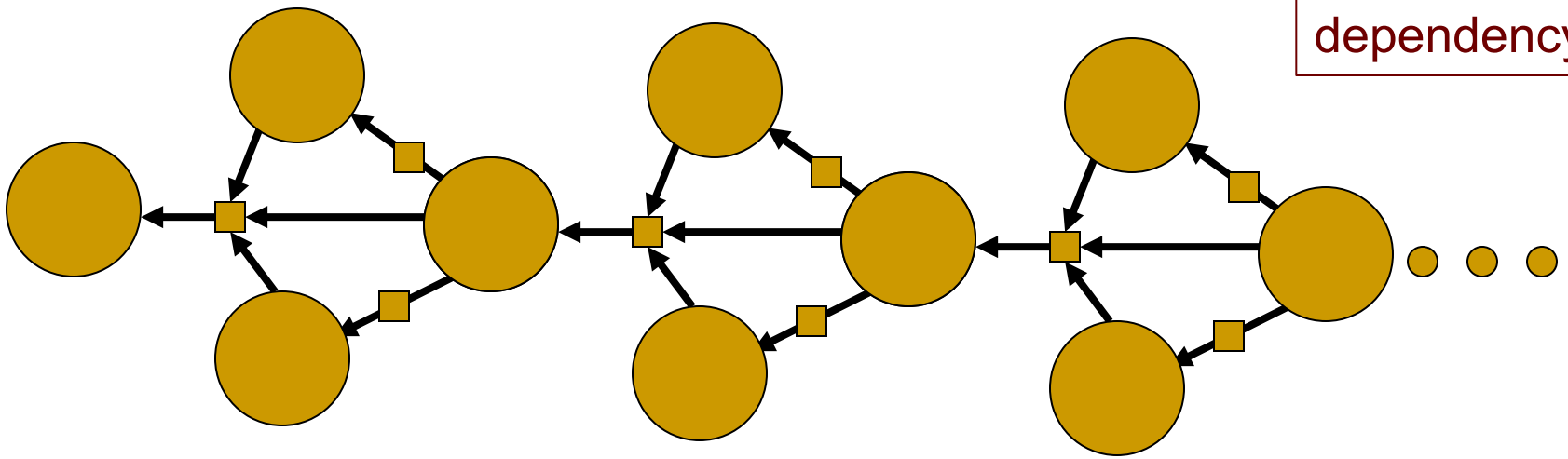
Naïve algorithm

```
For each variable v do
  evaluate(v)

evaluate(v):
  Parms = empty
  for each DepVar in v.dep do
     Parms += evaluate(DepVar)
  v.value = v.eqn(Parms)
  return v.value
```

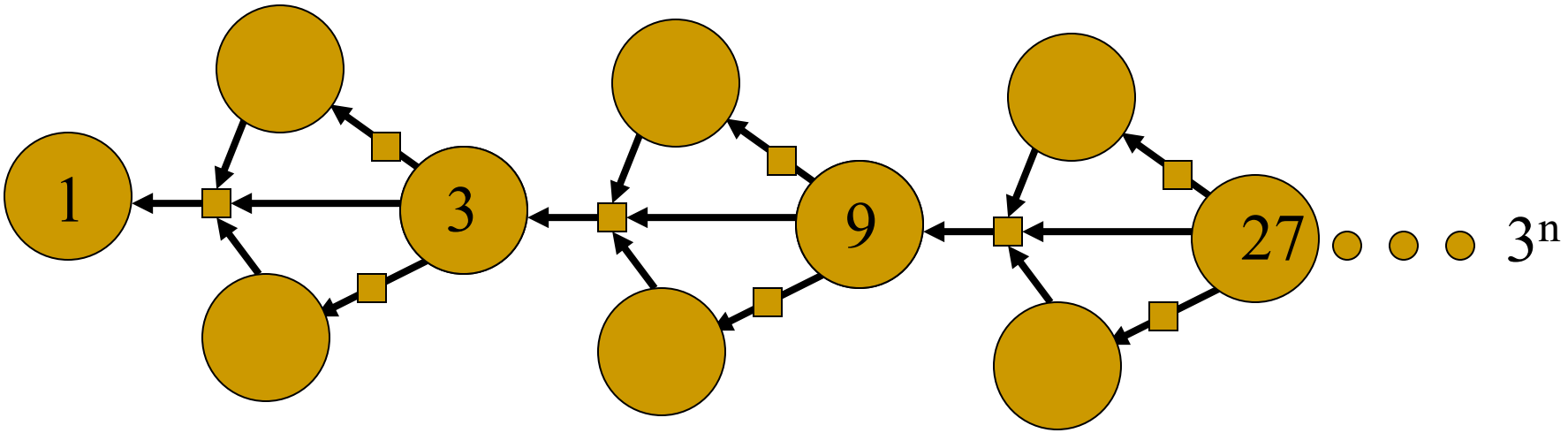
Why is this not a good plan?

Exponential Wasted Work

NOTE: These arrows are in the ***dataflow*** direction. Not dependency

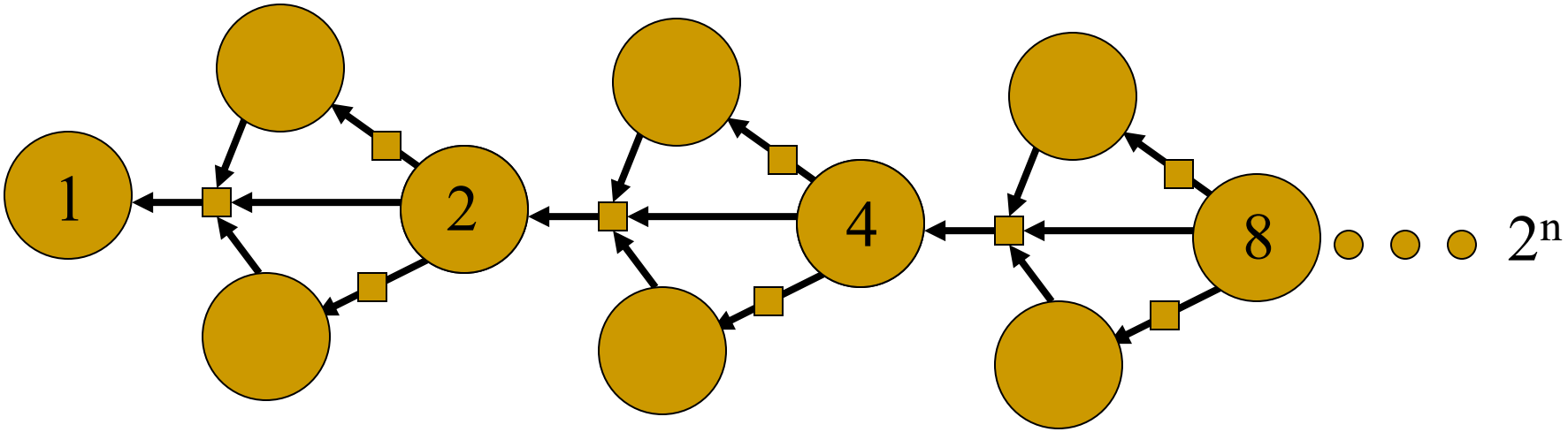


Exponential Wasted Work



Exponential Wasted Work

Breadth first does not fix this



No fixed order works for all graphs

Must respect topological ordering of graph (do in reverse topsort order)

Simple algorithm for one-way (Embed evaluation in topsort)

- After any change:

```
// reset all the marks
for each variable v do
  v.done = false

// make each var up-to-date
for each variable v do
  evaluate(v)
```

Simple algorithm for one-way

```
evaluate(V):
  if (!v.done)
    V.done = true
    Parms = empty
    for each DepVar in V.dep do
      Parms += evaluate(DepVar)
    V.value = V.eqn(Parms)
  return V.value
```

Still a lot of wasted work

- Typically only change small part of system, but this algorithm evaluates all variables every time

- Also evaluates variables even if nothing they depend on has changed, or system never needs value
  - e.g., with non-strict functions such as boolean ops and conditionals

An efficient incremental algorithm

- Add bookkeeping
  - For each variable: OODMark
    - “Out Of Date mark”
    - Indicates variable may be out of date with respect to its constraint
  - For each dependency edge: pending
    - Indicates that variable depended upon has changed, but value has not propagated across the edge

Part one (of two)

When variable (or constraint) changed, call MarkOOD() at point of change

```
MarkOOD(v):                                    [x]
  if !v.OODMark
    v.OODMark = true
    for each depV depending upon v do
      MarkOOD(depV)
```

Part one (of two)

When variable (or constraint) changed, call MarkOOD() at point of change

```
MarkOOD(v):
  if !v.OODMark
    v.OODMark = true
    for each depV depending upon v do
      MarkOOD(depV)
```

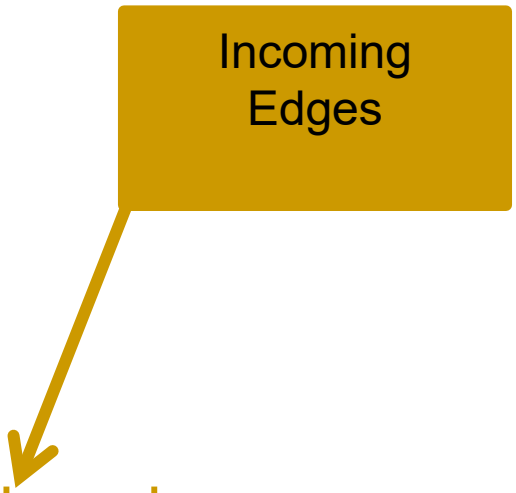
Outgoing Edges

Part 2: only evaluate variables when value requested (lazy eval)

```
Evaluate(v):
  if v.OODMark
    v.OODMark = false
    Parms = empty
    for each depVar in V.dep do
      Parms += Evaluate(depVar)
    UpdateIfPending(v,Parms)
  return v.value
```

Part 2: only evaluate variables when value requested (lazy eval)

```
Evaluate(v):
  if v.OODMark
    v.OODMark = false
    Parms = empty
    for each depVar in V.dep do
      Parms += Evaluate(depVar)
    UpdateIfPending(v,Parms)
  return v.value
```



Part 2: only evaluate variables when value requested (lazy eval)

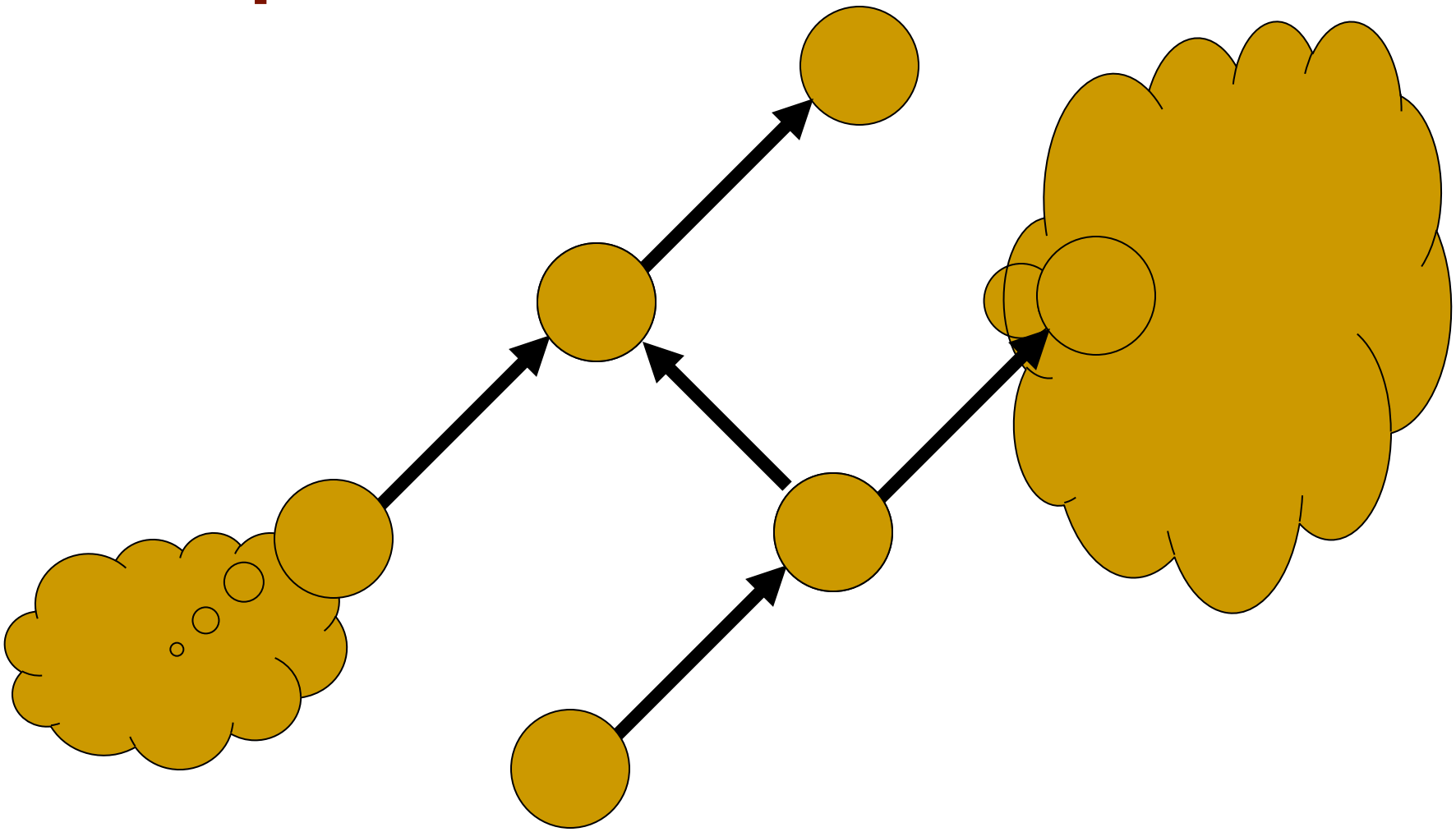
```
UpdateIfPending(v,Parms):
  pendingIn = false  //any incoming pending?
  For each incoming dep edge E do
    pendingIn |= E.pending
    E.pending = false

  if pendingIn
    newVal = V.eqn(Parms)        [*]
     if newval != v.value
       v.value = newval
       Foreach outgoing dependency edge D do
         D.pending = true
```

Part 2: only evaluate variables when value requested (lazy eval)

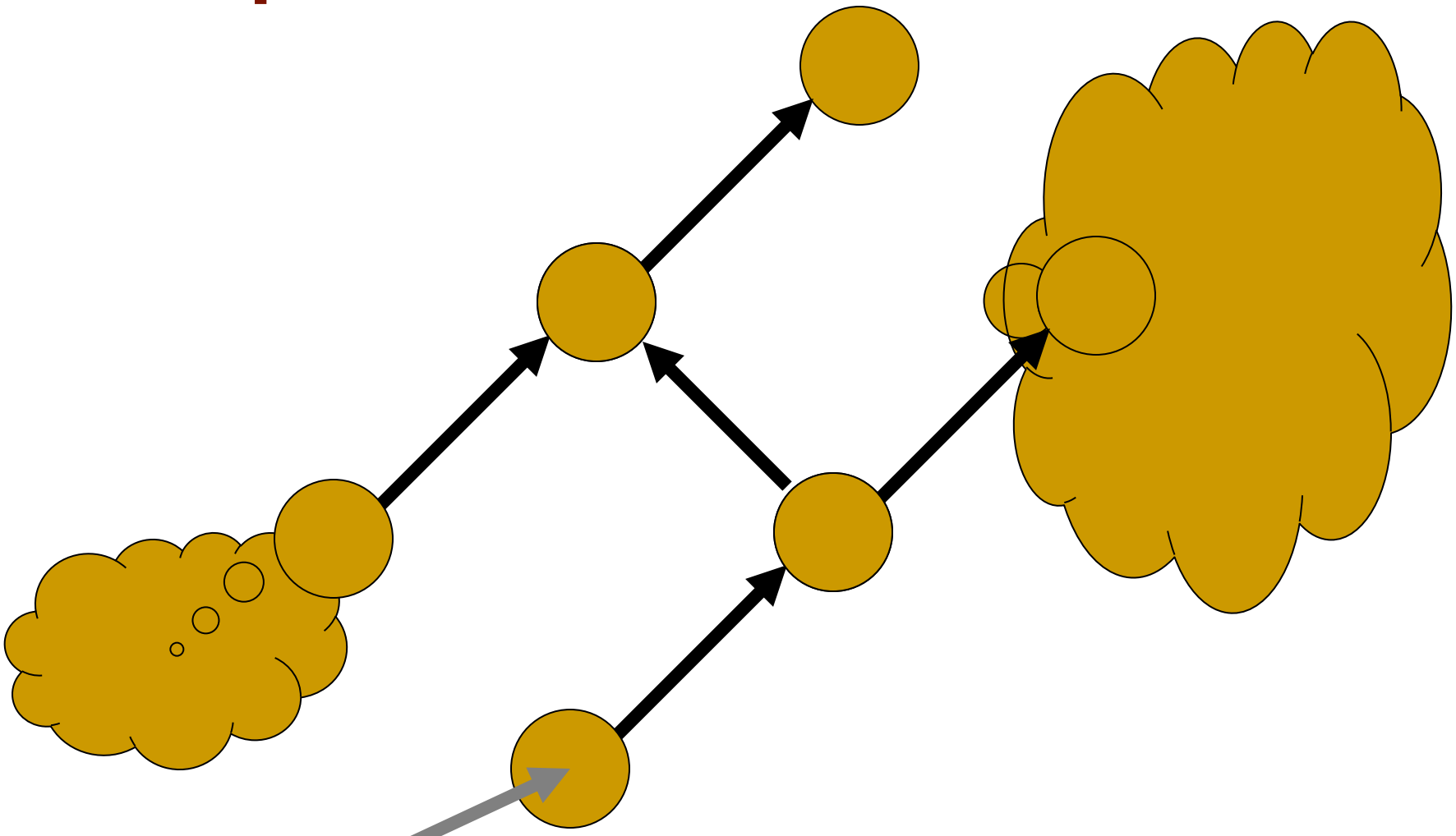
```
UpdateIfPending(v,Parms):
  pendingIn = false  //any incoming pending?
  For each incoming dep edge E do
    pendingIn |= E.pending
    E.pending = false

  if pendingIn
    newVal = V.eqn(Parms)          [*]
     if newval != v.value
       v.value = newval
       Foreach outgoing dependency edge D do
         D.pending = true
```

Can do lazy evaluation here

Example

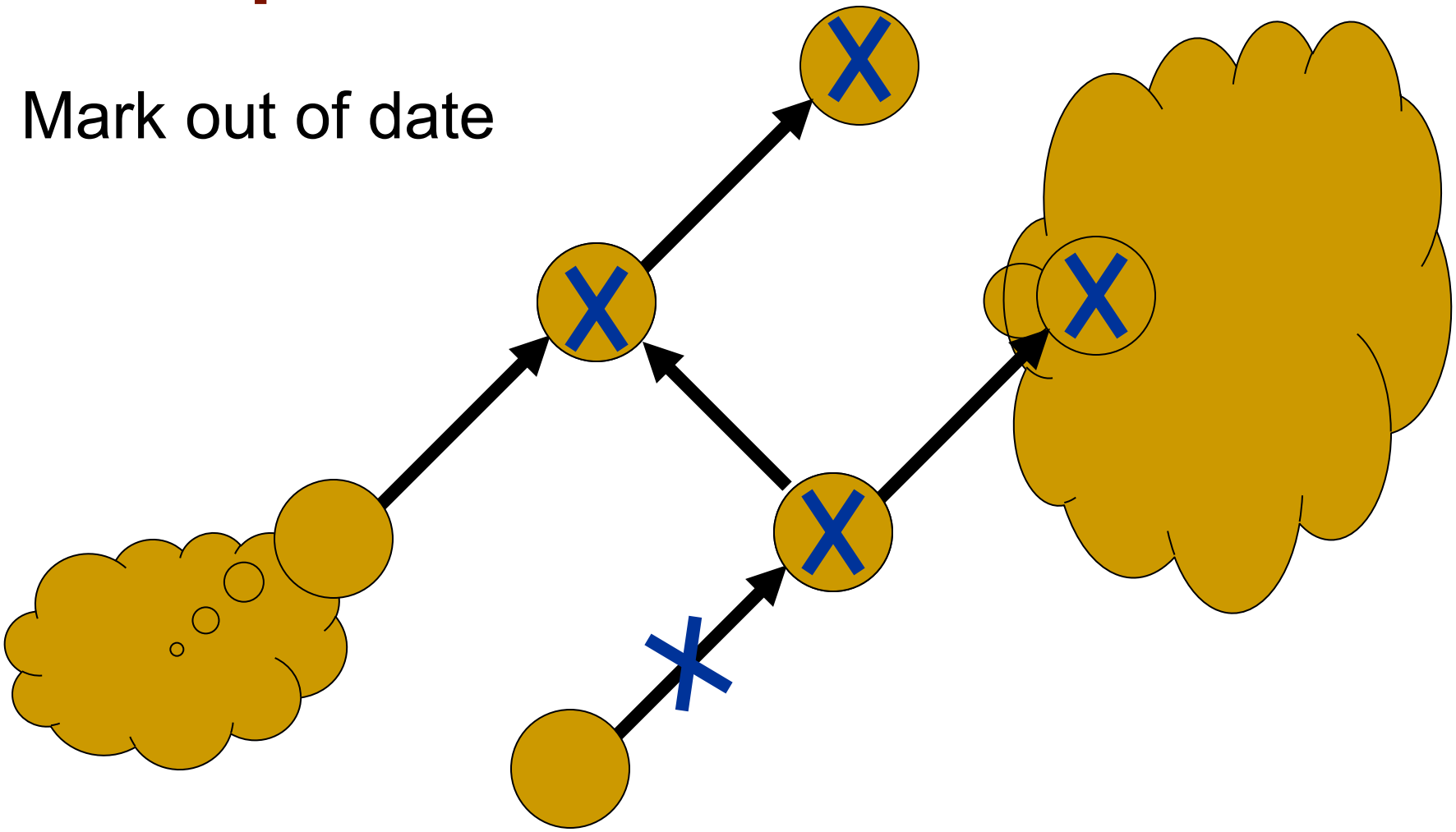


Example



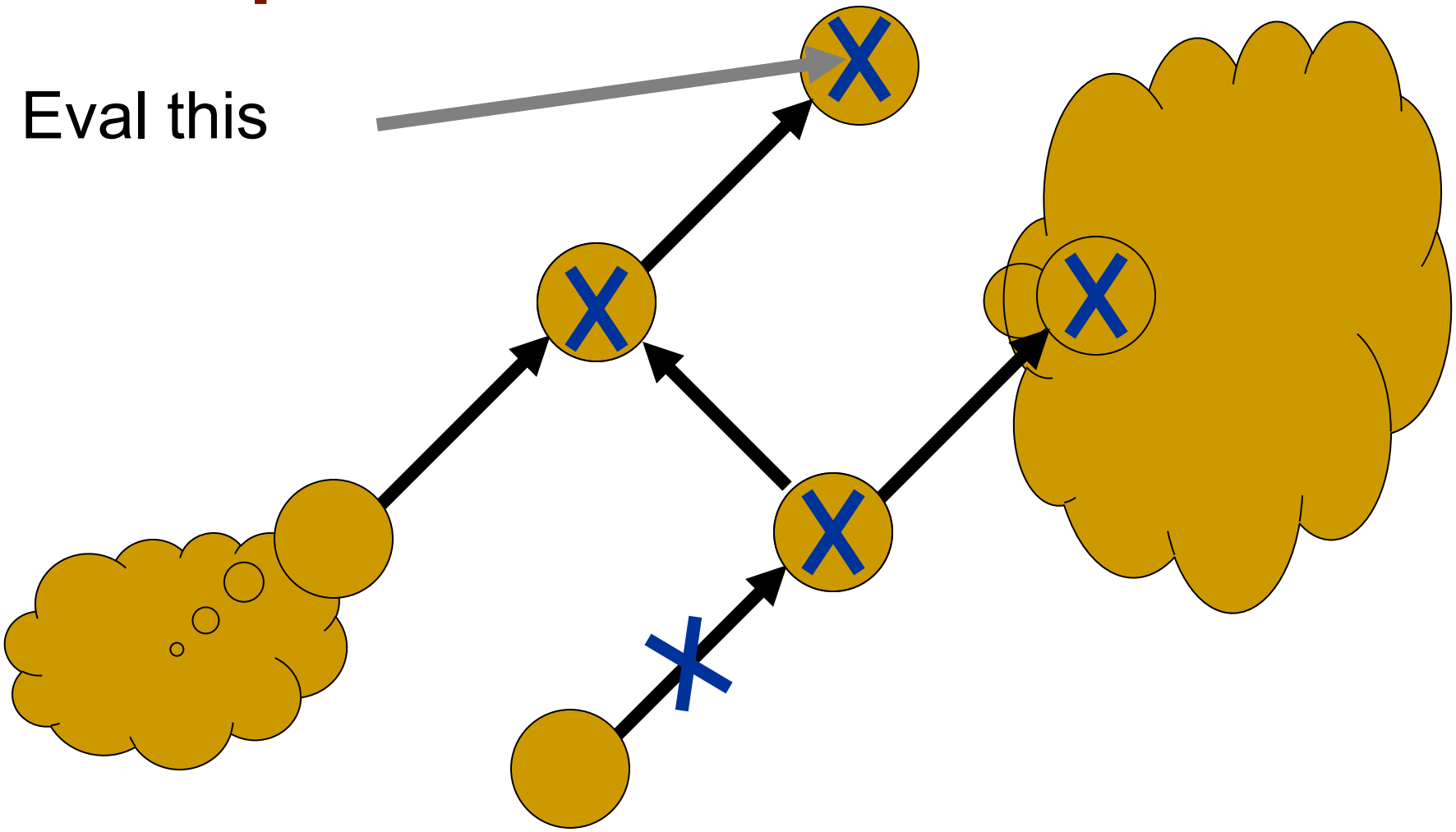
Example

Mark out of date

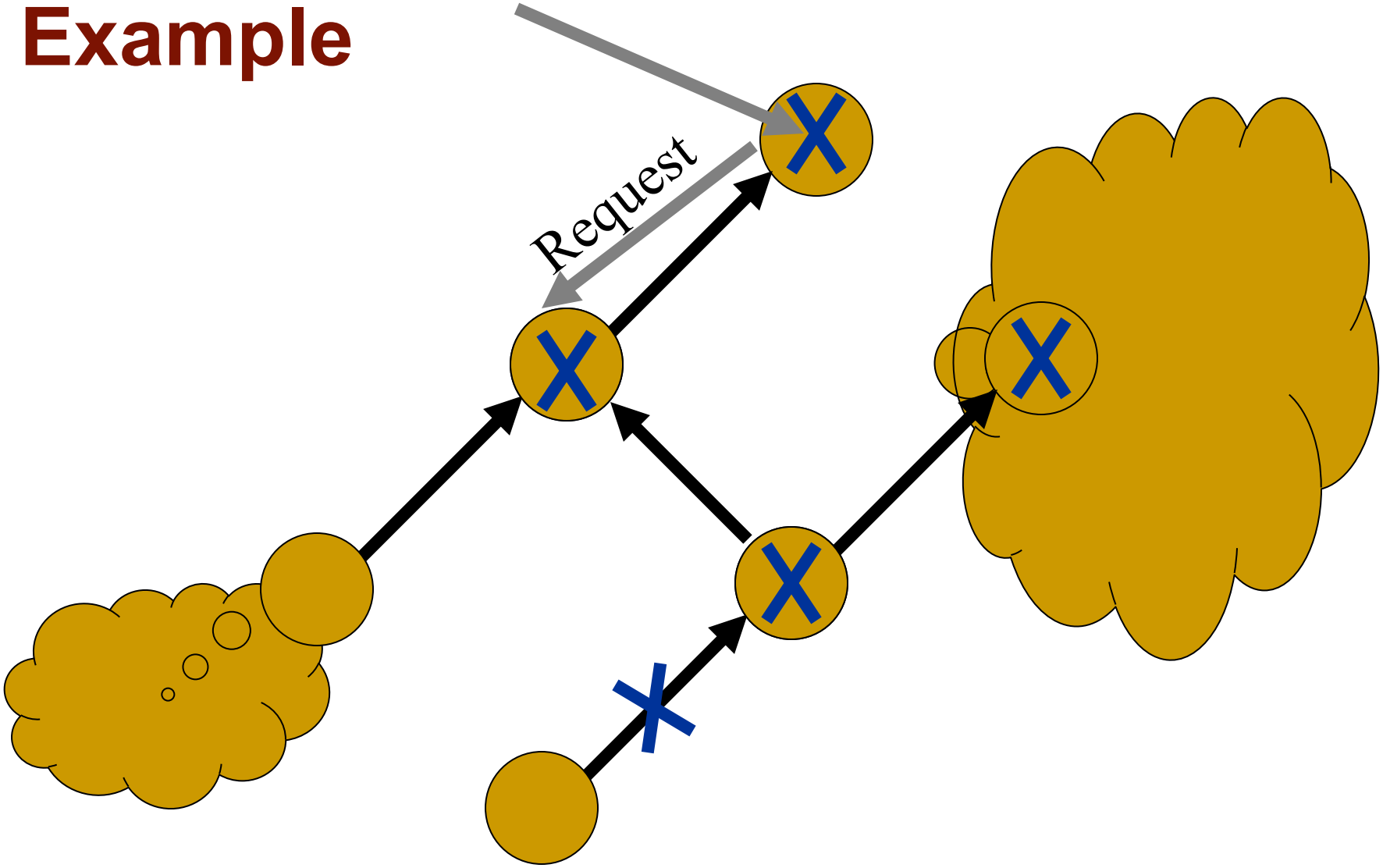


Example

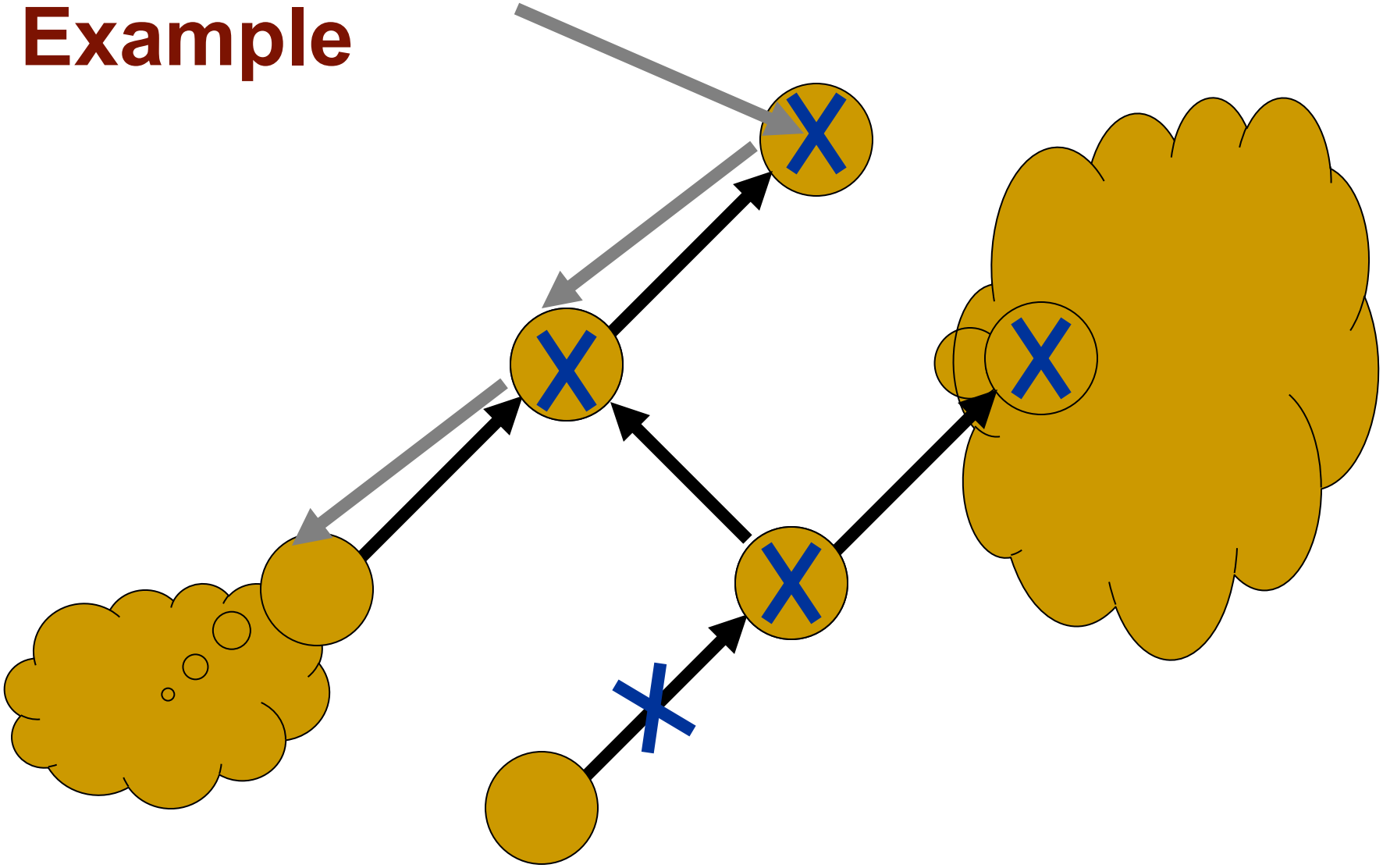
Eval this



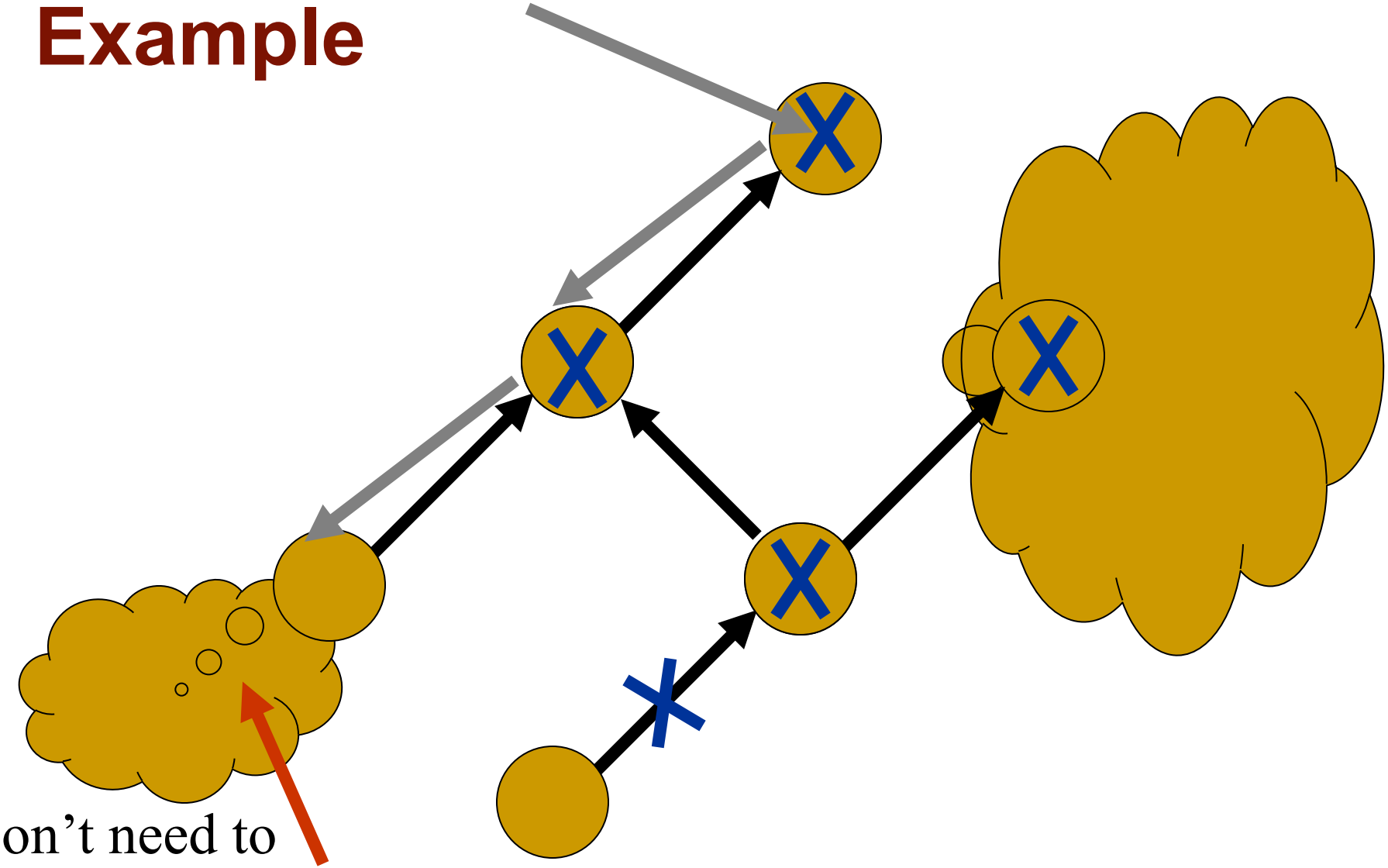
Example



Example

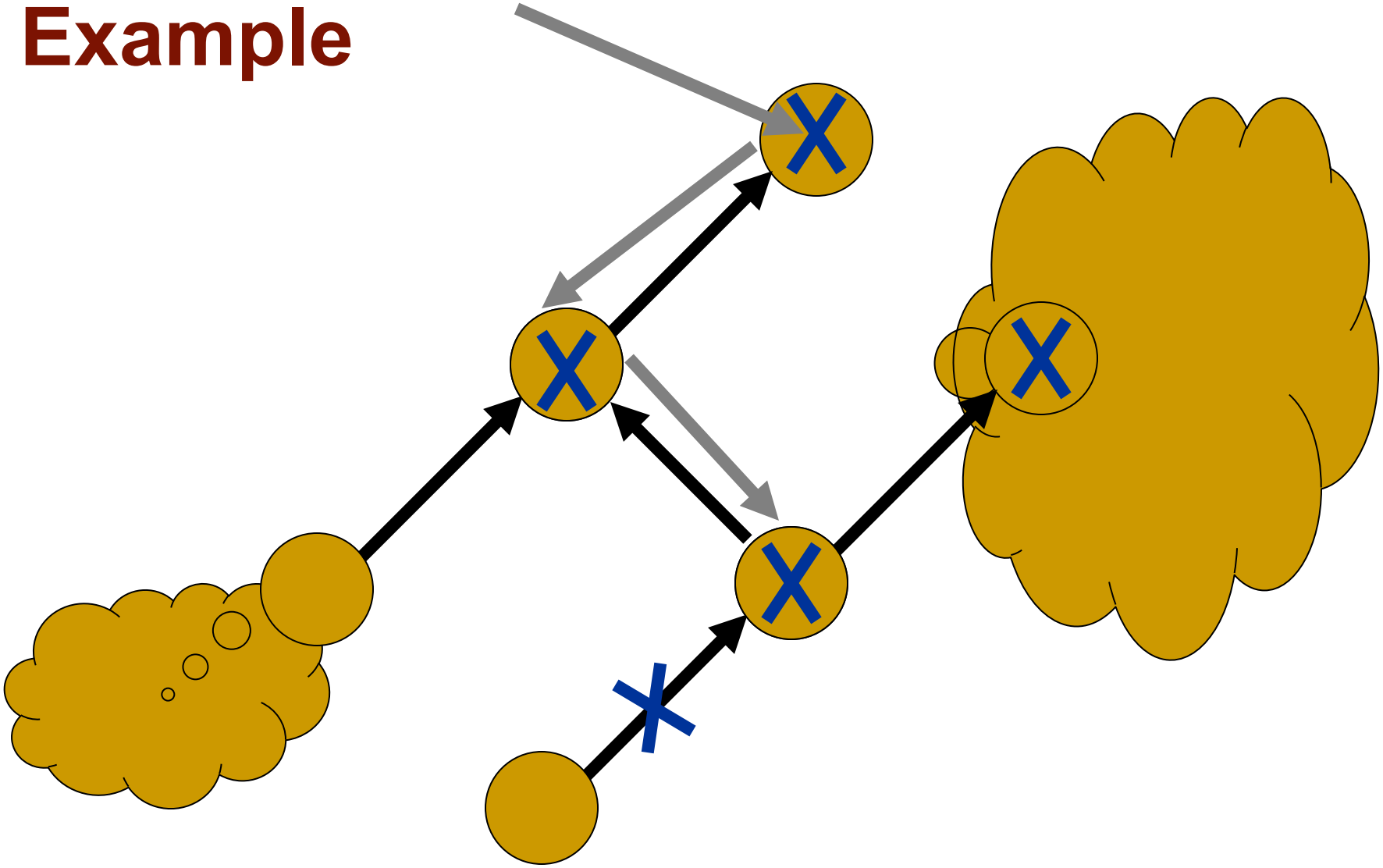


Example

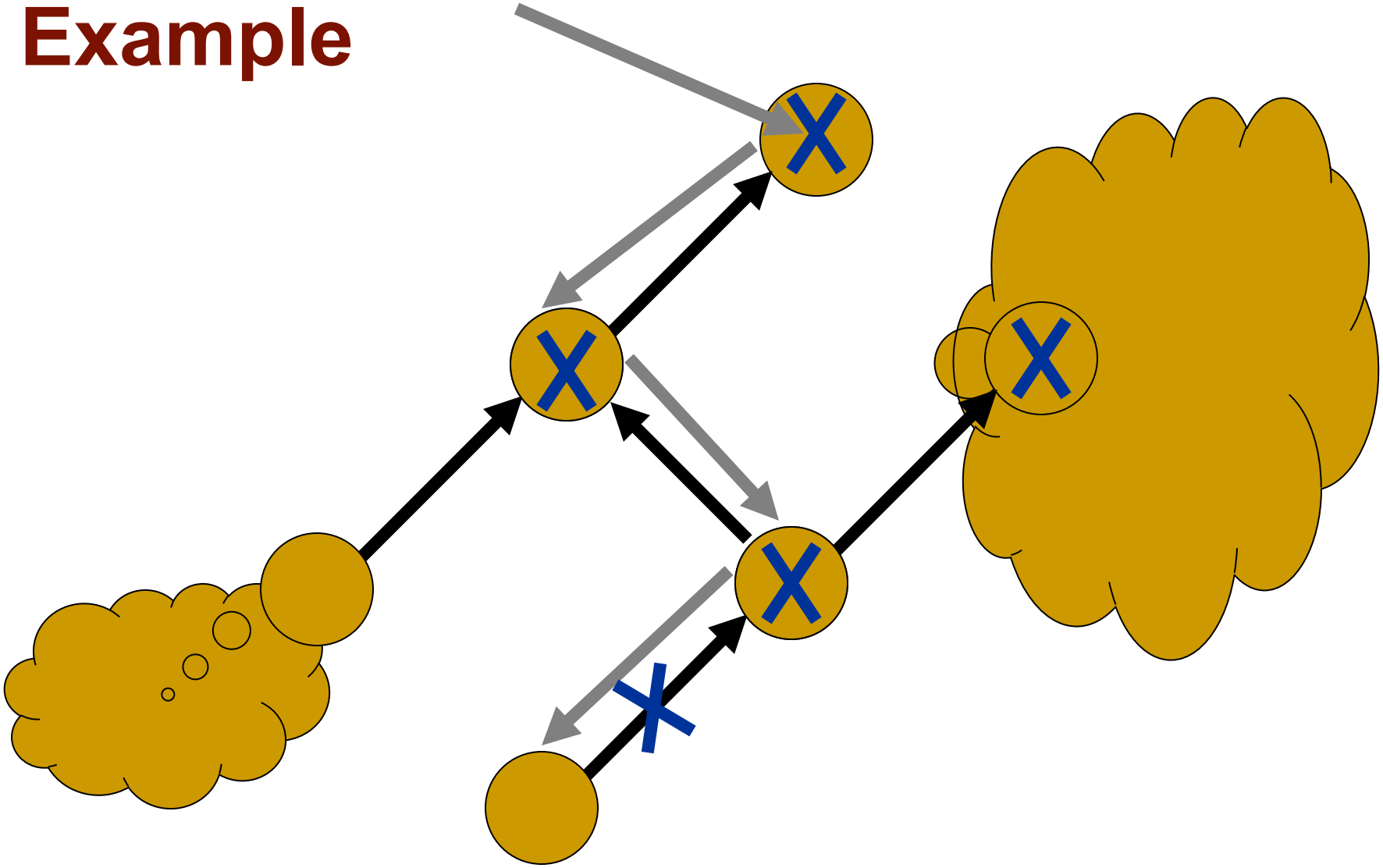


Don’t need to
eval any of these! (Not out-of-date)

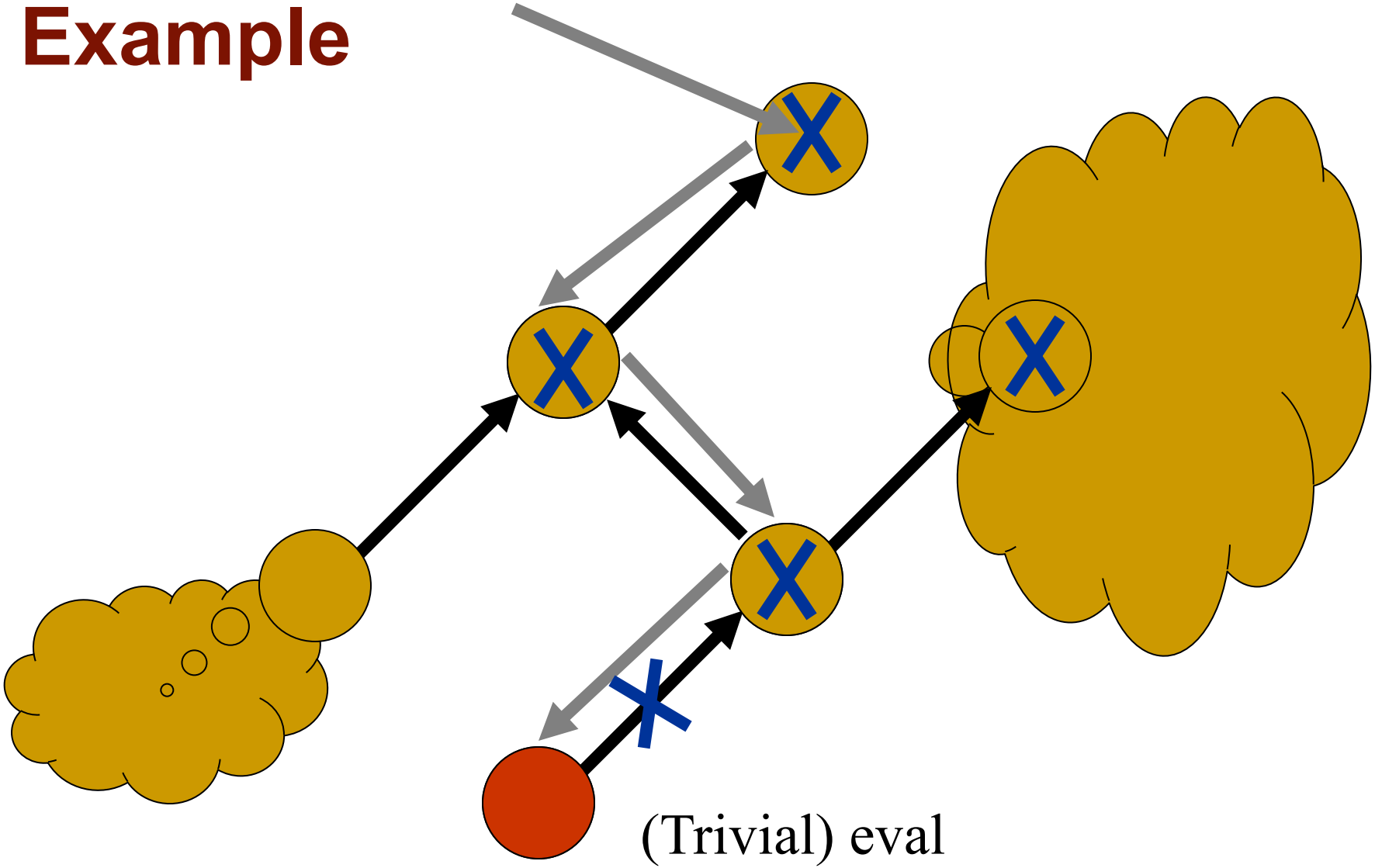
Example



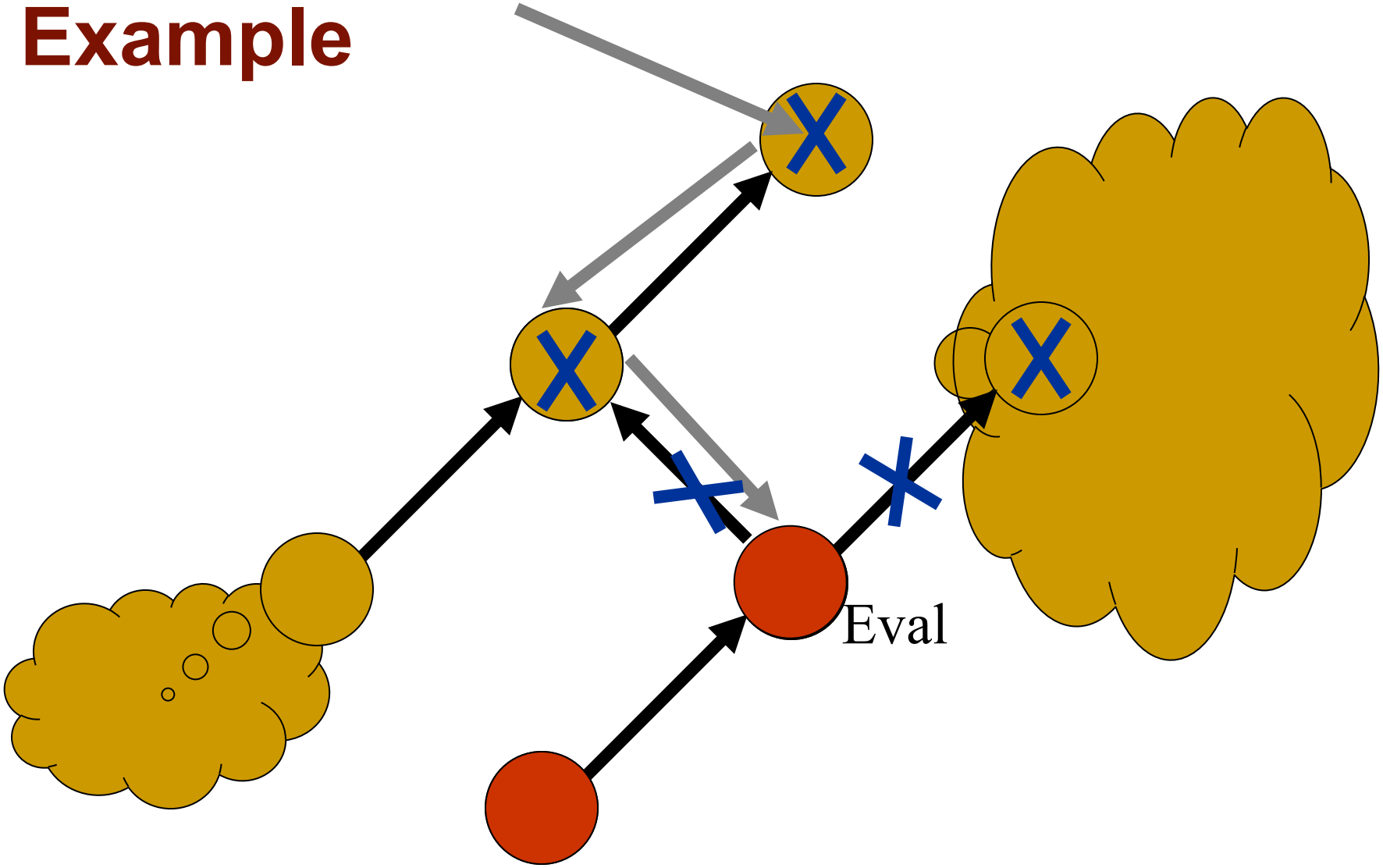
Example



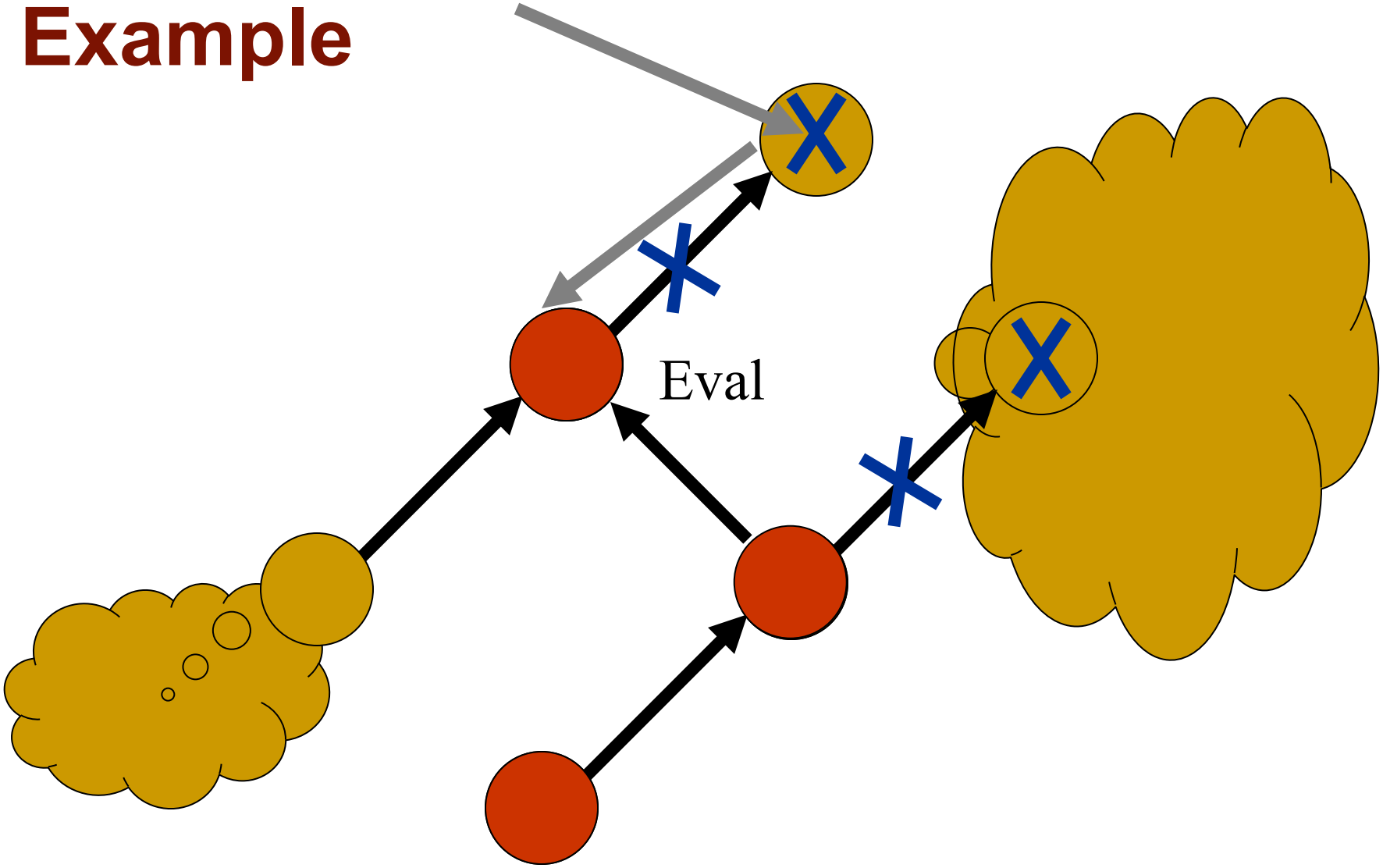
Example



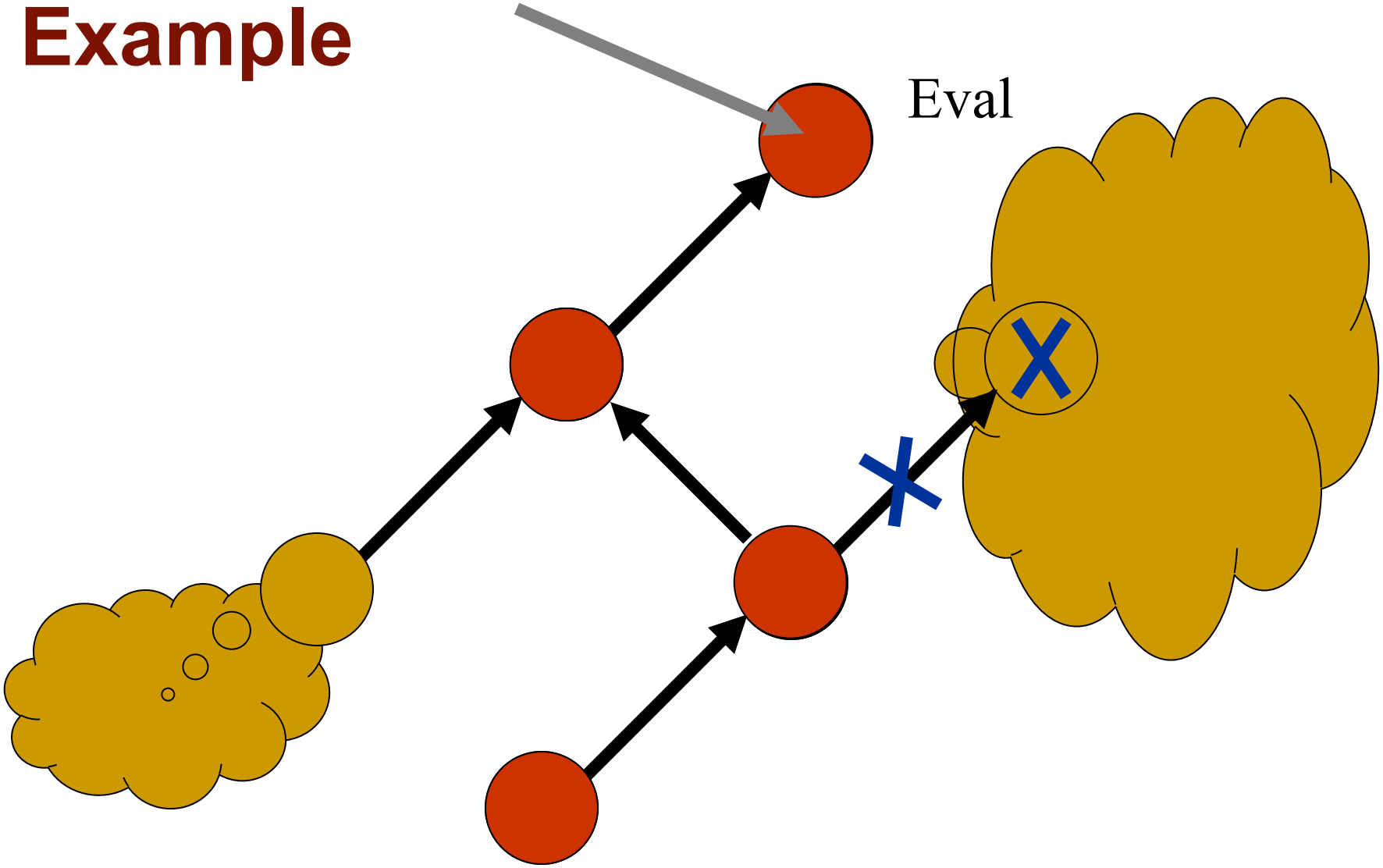
Example



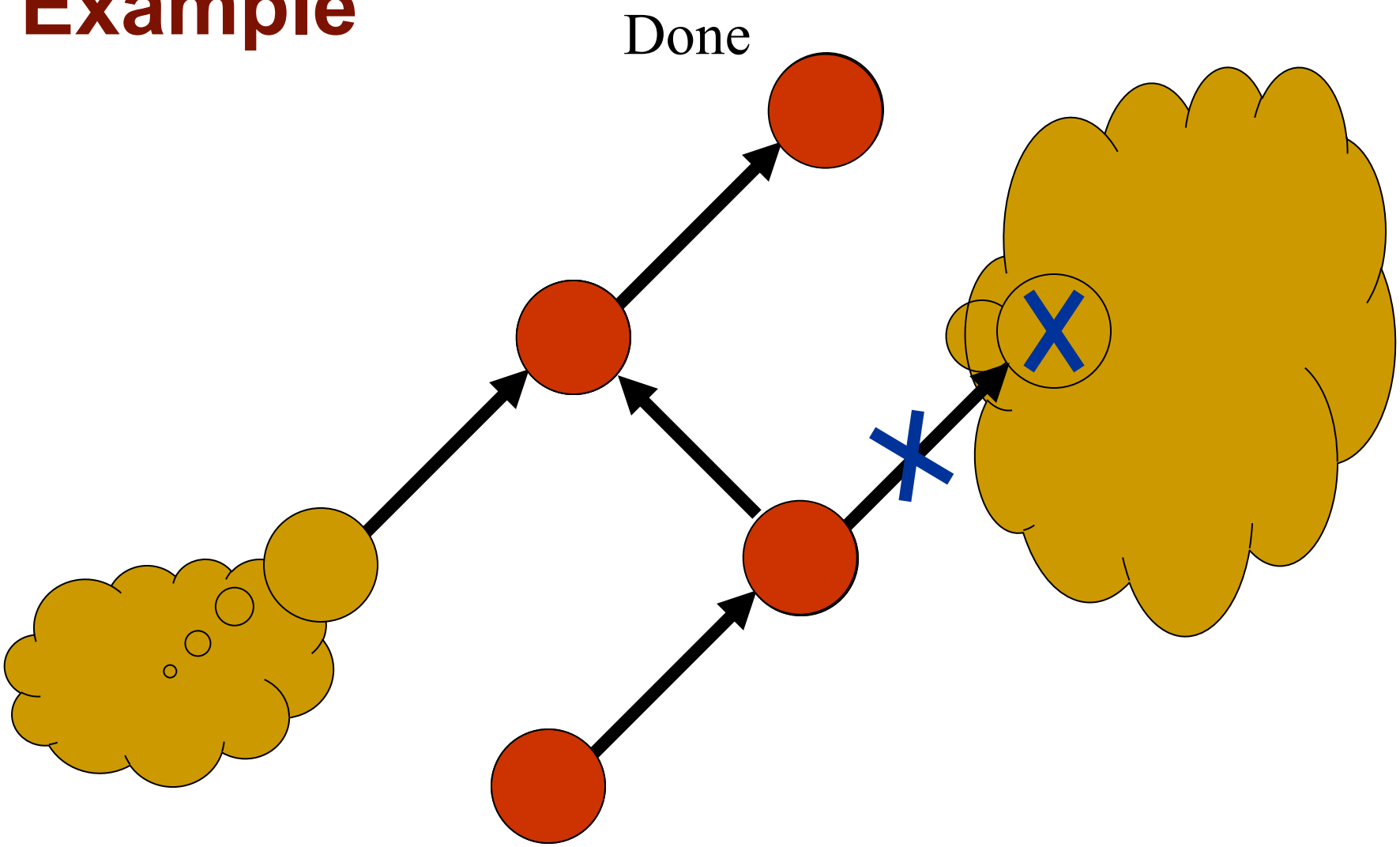
Example



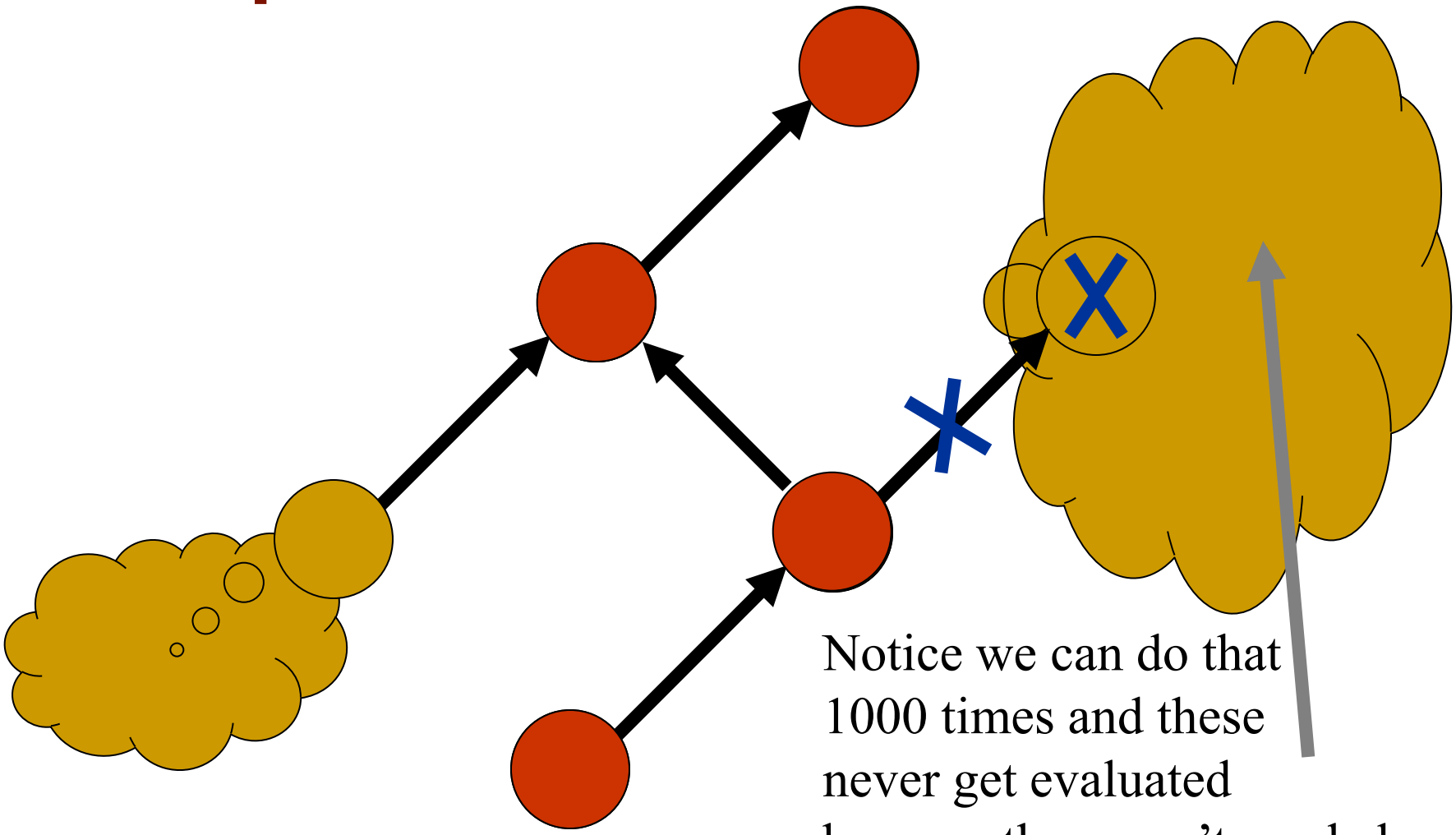
Example



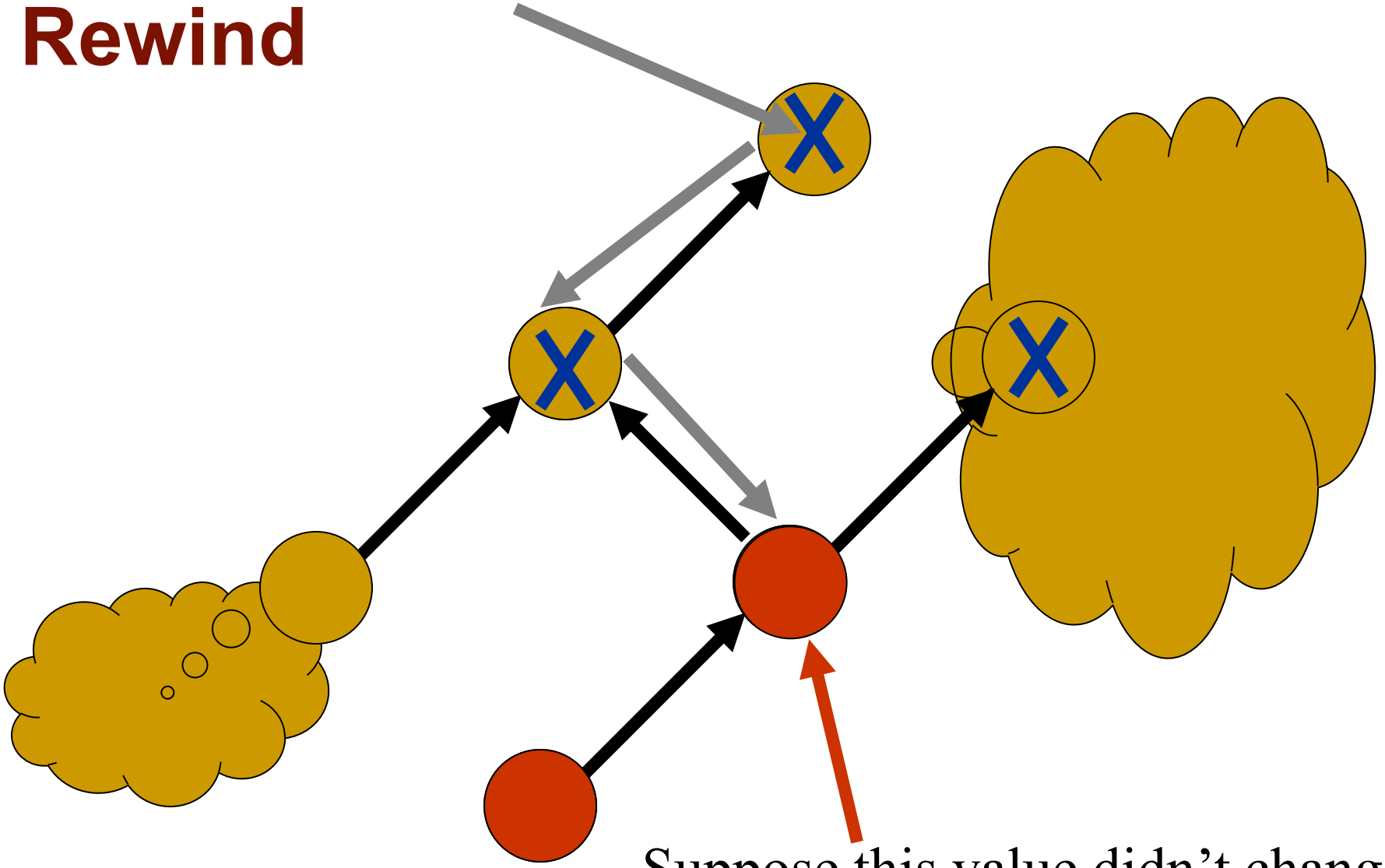
Example



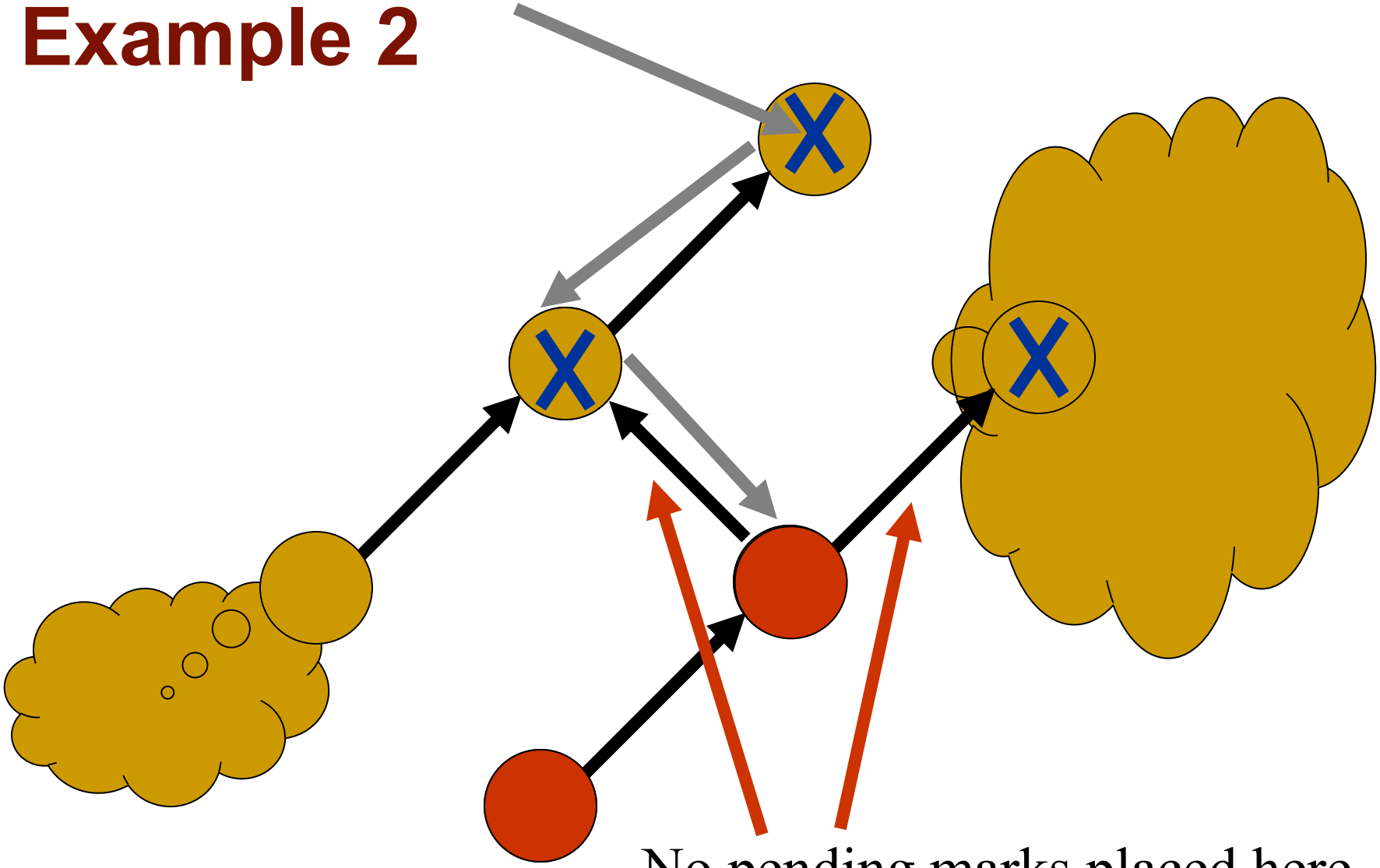
Example



Rewind

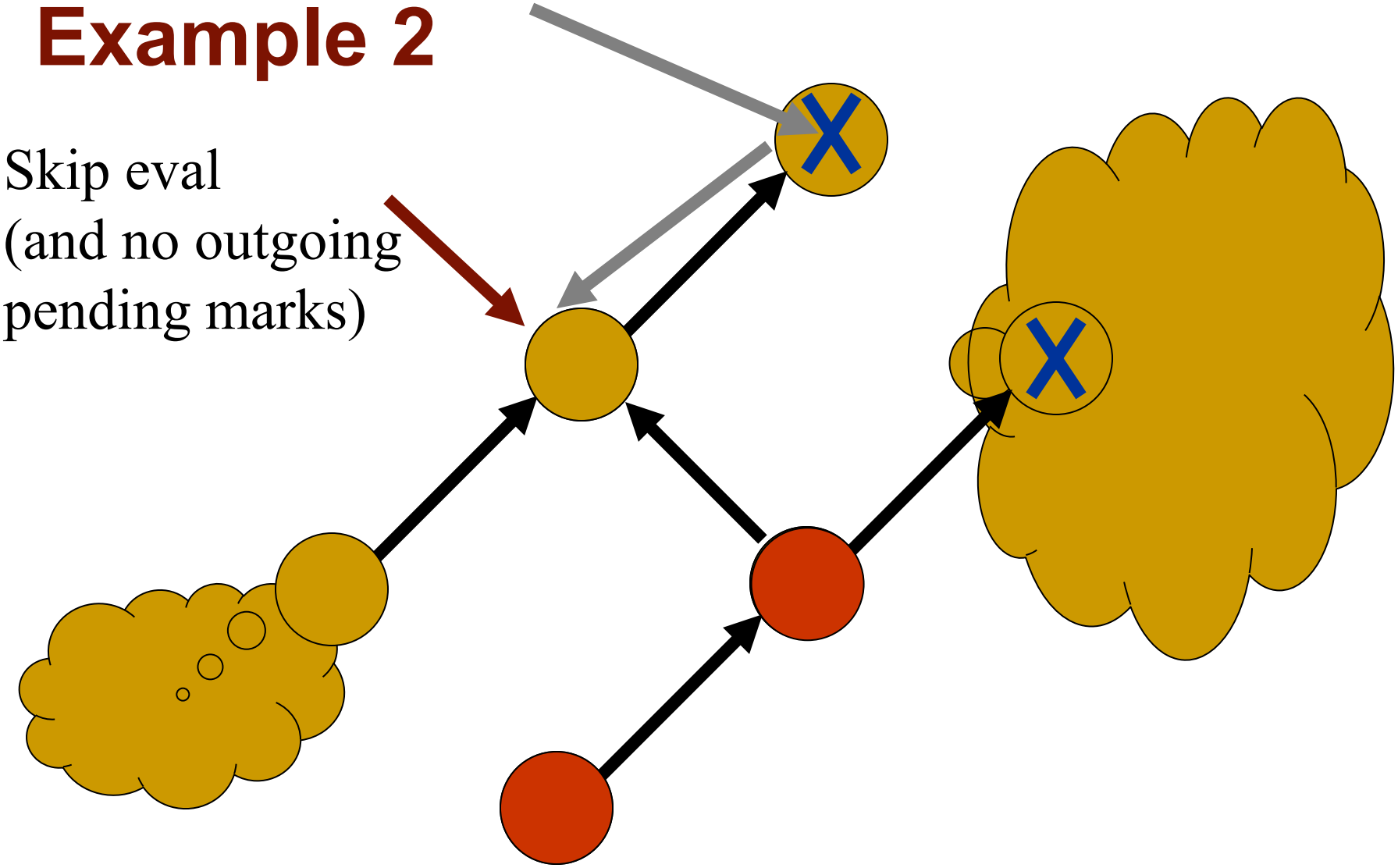


Example 2

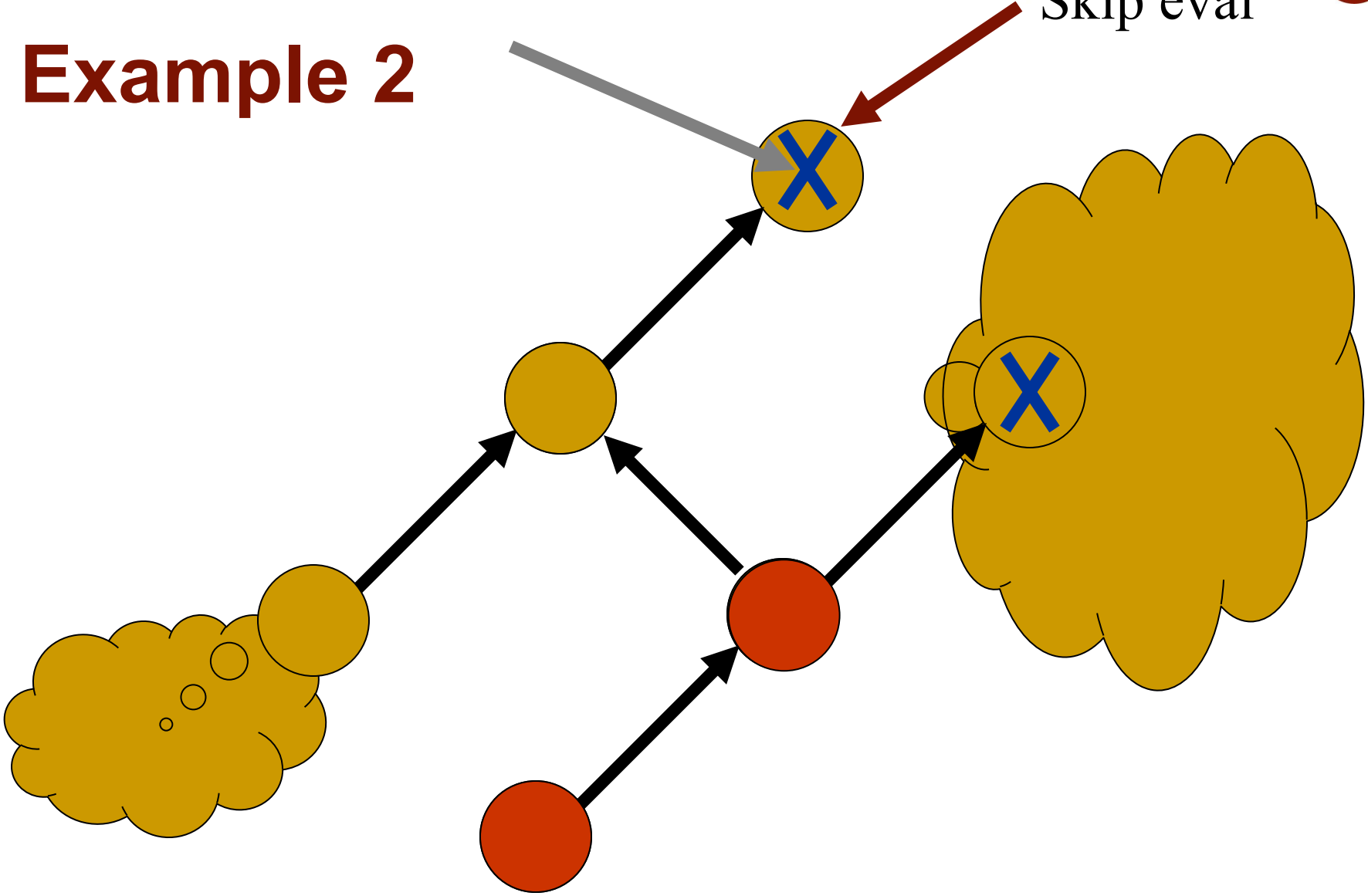


Example 2

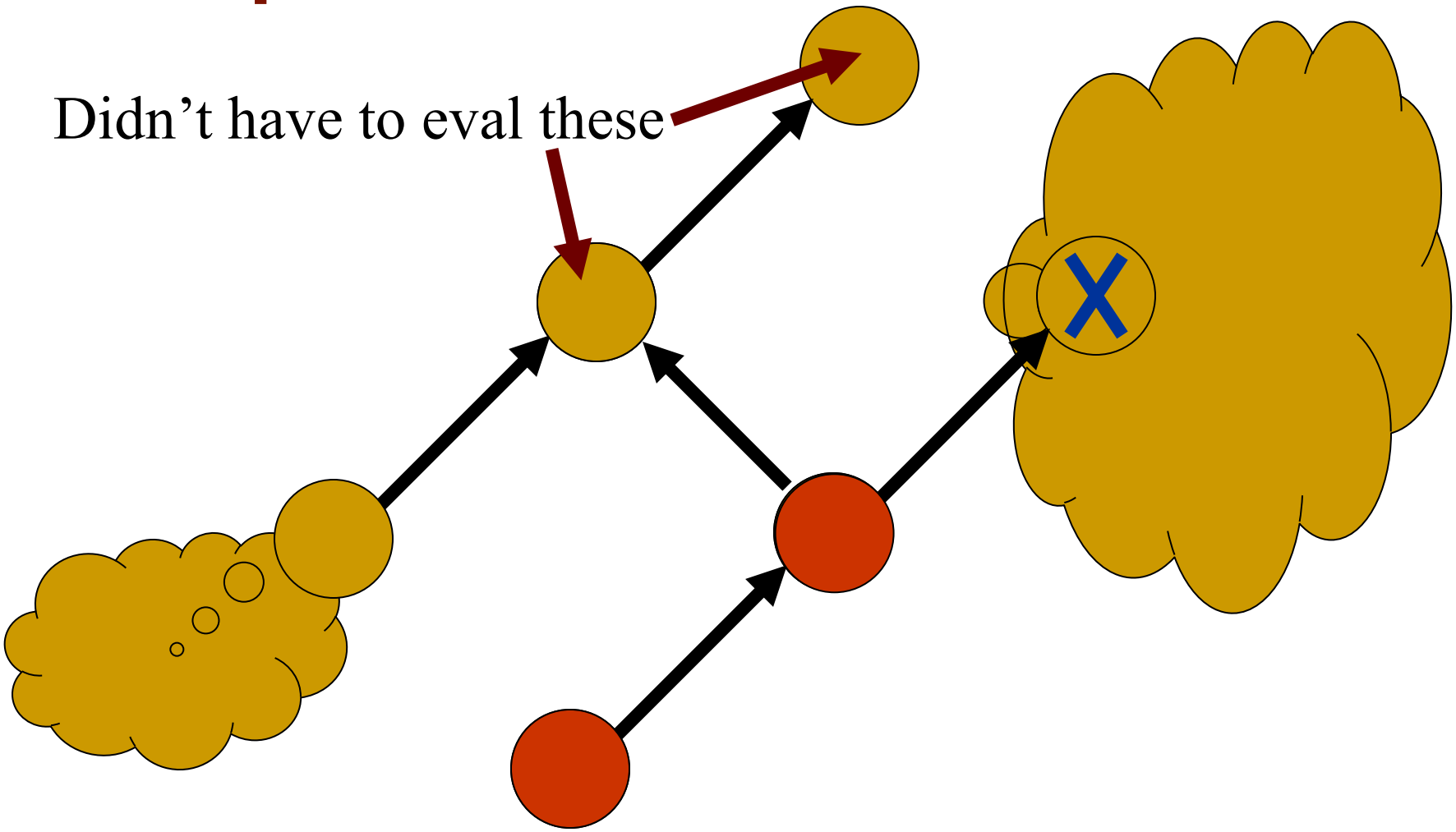
Skip eval
(and no outgoing
pending marks)



Example 2



Example 2



Algorithm is “partially optimal”

- Optimal in set of equations evaluated [*]
  - Under fairly strong assumptions
- Does non-optimal total work [x]
  - “Touches” more things than optimal set during Mark_OOD phase
    - Fortunately simplest / fastest part
  - Very close to theoretical lower bound
  - No better algorithm known

Good asymptotic result, but also very practical

- Minimal amount of bookkeeping
  - Simple and statically allocated
  - Only local information
- Operations are simple
- Also has very simple extension to handling pointers and dynamic dependencies

Multi-way implementation

- Use a “planner” algorithm to assign a direction to each undirected edge of dependency graph
- Now have a one-way problem

The DeltaBlue incremental planning algorithm

- Assume “constraint hierarchies”
  - Strengths of constraints
  - Important to allow more control when over or under constrained
    - Force all to be over constrained, then relax weakest constraints
    - Substantially improves predictability
- Restriction: acyclic (undirected) dependency graphs only

A plan is a set of edge directions

- Assume we have multiple methods for enforcing a constraint
  - One per (output) variable
  - Picking method sets edge directions
- Given existing plan and change to constraints, find a new plan

Finding a new plan

- For added constraints
  - May need to break a weaker constraint (somewhere) to enforce new constraint
- For removed constraints
  - May have weaker unenforced constraints that can now be satisfied

Finding possible constraints to break when adding a new one

- For some variable referenced by new constraint
  - Find an undirected path from var to a variable constrained by a weaker constraint (if any)
  - Turn edges around on that path
  - Break the weaker constraint

Key to finding path: "Walkabout Strengths"

- Walkabout strength of variable indicates weakest constraint "upstream" from that variable
  - Weakest constraint that could be revoked to allow that variable to be controlled by a different constraint

Walkabout strength

- Walkabout strength of var V currently defined by method M of constraint C is:
  - Min of C.strength and walkabout strengths of variables providing input to M

DeltaBlue planning

- Given WASs of all vars
  - (WalkAbout Strength)
- To add a constraint C:
  - Find method of C whose output var has weakest WAS and is weaker than C
    - If none, constraint can't be satisfied
  - Revoke constraint currently defining that var
  - Attempt to reestablish that constraint recursively
    - Will follow weakest WAS
  - Update WASs as we recurse

DeltaBlue Planning

- To remove a constraint C
  - Update all downstream WASs
  - Collect all unenforced weaker constraints along that path
  - Attempt to add each of them (in strength order)

DeltaBlue Evaluation

- A DeltaBlue plan establishes an evaluation direction on each undirected dependency edge
- Based on those directions, can then use a one-way algorithm for actual evaluation

References

- Optimal one-way algorithm
  http://doi.acm.org/10.1145/117009.117012
  Note: constraint graph formulated differently
  - Edges in the other direction
  - No nodes for functions (not bipartite graph)

- DeltaBlue
  http://doi.acm.org/10.1145/76372.77531