Lecture 26:
Constraints and Data Bindings

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

Fall, 2022

Logistics

- Last regular lecture!
- Student group presentations Thursday+Friday
  - Everyone is expected to attend both in person
- Please fill out the class questionnaire: https://www.surveymonkey.com/r/SSUI2022Fall-Final

Constraints

- Relationships defined once and maintained by the system
- Useful for keeping parts of the graphics together.
- Also for passing values around
- Typically expressed as arithmetic or code relationships among variables.
  - Variables are often the properties of objects (left, color)
- Types:
  - "Dataflow" constraints; Choices:
    - Single-Output vs. Multi-output
    - Types: One-way, Multi-way, Simultaneous equations, Incremental, Special purpose
    - Cycles: supported or not
  - Others: AI systems, scheduling systems, etc.

Historical Note: "Active Values"

- Old Lisp systems had active values
  - Attach procedures to be called when changed
- Similar to today's "Listeners" or "Observer pattern"
- Like the "inverse" of constraints
  - Procedures are attached to values which change instead of values where needed
  - Push vs. Pull
- Inefficient because all downstream values are re-evaluated, possibly many times
  - E.g., when x and y values change

Important Historical Constraint Systems

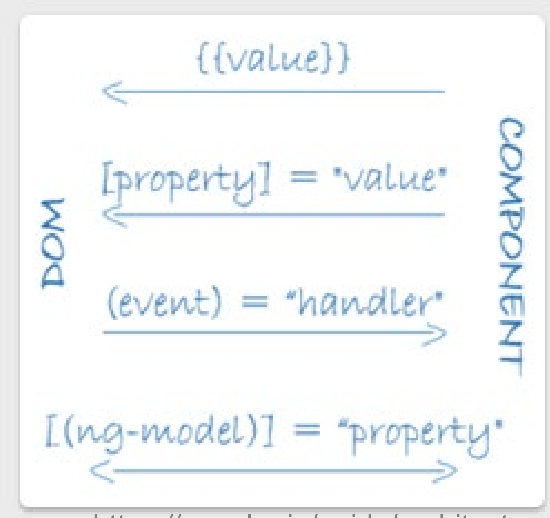
- Alan Borning's ThingLab (1979)
- Spreadsheets (~1979)
- Peridot (1987) (Myers)
- Garnet & Amulet (1989, 1994) (Myers)
  - Graphics *and* "data bindings"
- DeltaBlue (1990) (Freemen-Benson)
  - SkyBlue (1994) (Michael Sannella)
- subarctic (Hudson) (1991)
- Gleicher's (1993)
- …

Some Constraint Systems Today

- Apple constraints for "Auto Layout"
- Toolkit and windows "layout managers"/"geometry managers" (lecture 10)
- "data bindings"
  - Usually one-to-one two-way connections
  - Adobe Flex, AngularJS
- Google's AngularJS (before v2)
- Most AutoDesk (CAD) products, e.g., Fusion 360 for 2D & geometric
- Ember. http://emberjs.com/
  - MVC, "Computed Values" of properties
- KnockoutJS. http://knockoutjs.com/
  - "Declarative Bindings", "Dependency Tracking"
- Research: Stephen Oney's ConstraintJS https://from.so/ (2012)

Angular Data Bindings

- Tie DOM properties to other values
- Can be one-way or two-way
  - Use [] to bind from source to view.
  - Use () to bind from view to source.
  - Use [()] to bind in a two way sequence of view to source to view.



https://angular.io/guide/architecture-components#data-binding

Type	Syntax	Category

Interpolation
Property
Attribute
Class
Style
	`{{expression}}`
	`[target]="expression"`
	`bind-target="expression"`
		One-way from data source to view target

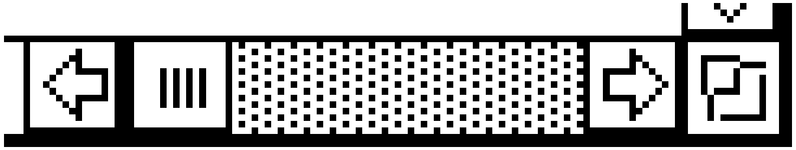
Event	`(target)="statement"` `on-target="statement"`	One-way from view target to data source
Two-way	`[(target)]="expression"` `bindon-target="expression"`	Two-way

https://angular.io/guide/binding-syntax

One Way Constraints

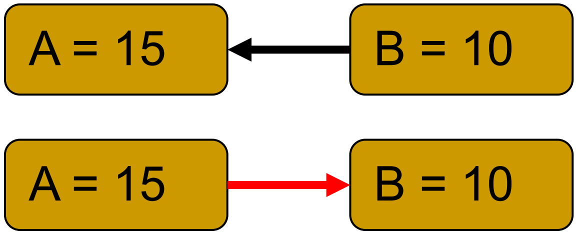
- Simplest form of constraints
- D = F(I1, I2, ... In)
- Often called *formulas* since like spreadsheets
- Can be other dependencies on D

CurrentSliderVal = mouse.X - scrollbar.left
scrollbar.left = window.left + 200
scrollbar.visible = window.has_focus



Data flow graph

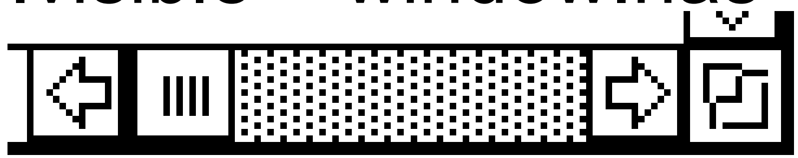
- Nodes for variables (values) grouped into objects
- Lines for data flow for the constraints
  - Reverse direction of lines for "dependencies"
  - E.g., A = B+5
    - B's value **flows to** A
    - A's value **depends on** B

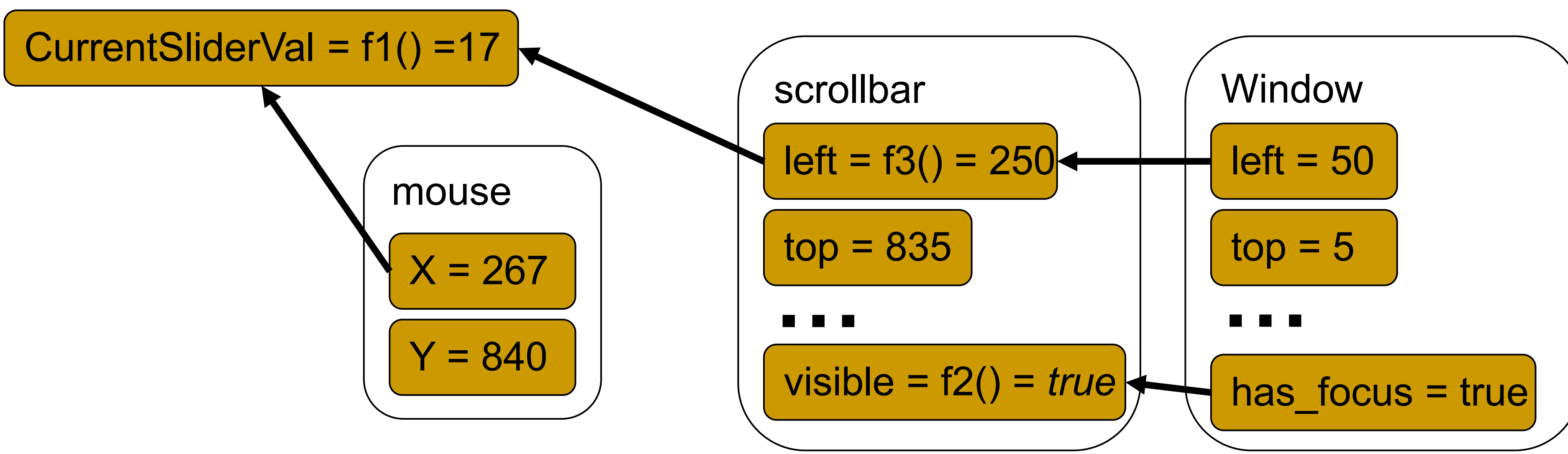


- Often need back-pointers too to clean up when change

One Way Constraints

CurrentSliderVal = mouse.X - scrollbar.left
scrollbar.left = window.left + 200
scrollbar.visible = window.has_focus





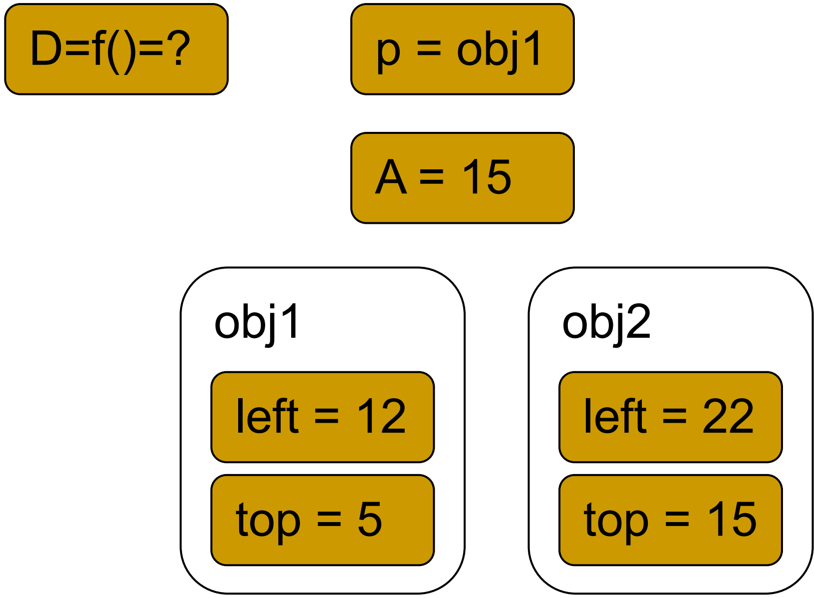
One Way Constraints, cont.

- Not just for numbers: mycolor = x.color
- Implementations:
  1. Just re-evaluate all required equations every time a value is requested
     - least storage, least overhead
     - Equations may be re-evaluated many times when not changed. (e.g, scrollbar.left when mouse moves)
     - cycles:
       file_position = F1(scrollbar.Val)
       scrollbar.Val = F2(file_position)
     - Objects may jitter – change X and then change Y
     - Cannot detect when values change (to optimize redraw)
  2. More efficient algorithms are available

Garnet / Amulet Constraint Solving

- Default: one-way, data flow constraints with variables in the dependencies, support for cycles, and multiple changes before solving
  - Efficient enough for ubiquitous use
  - Garnet text button widget contained 43 constraints internally, and the Lapidary graphical interface builder contained 16,700 constraints
- Also can bring in alternative solvers
  - Brad Vander Zanden's multi-way solver [Vander Zanden 1996]
  - "Animation Constraints" [Myers 1996]
- Snippets of video for Garnet and Amulet constraints

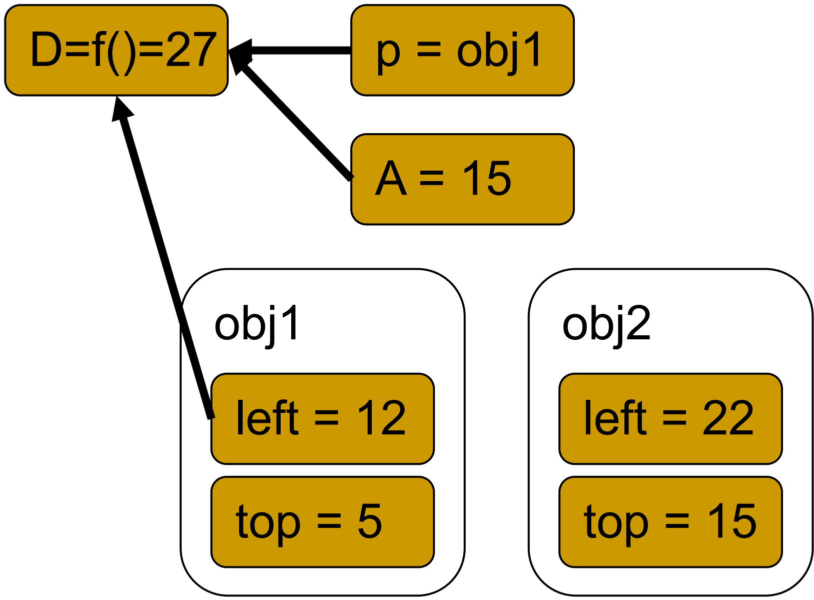
Garnet / Amulet Default Algorithm



- Variables in the dependencies
  - Example: D = p^.left + A
  - Important innovation in Garnet we invented, now ubiquitous
  - Supports feedback objects
    - outlineRect.left = selectedObject^.left …

      circle1.object_over = rect34
      circle1.left = self.object_over.right + 10
  - Supports loops: D = Max(components^)
  - Only evaluates needed part of conditionals
    width = if otherpart.value > tolerance
      then *expensive computation*
      else otherpart.width
  - Requires the dependencies be dynamically determined

Garnet / Amulet Default Algorithm

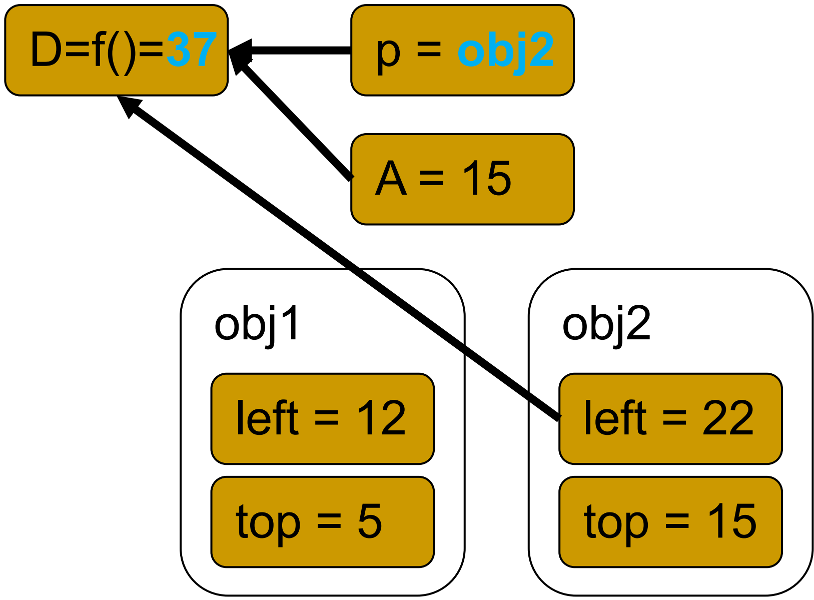


- Variables in the dependencies
  - Example: D = p^.left + A
  - Important innovation in Garnet we invented, now ubiquitous
  - Supports feedback objects
    - outlineRect.left = selectedObject^.left …

      circle1.object_over = rect34
      circle1.left = self.object_over.right + 10
  - Supports loops: D = Max(components^)
  - Only evaluates needed part of conditionals
    width = if otherpart.value > tolerance
       then *expensive computation*
       else otherpart.width
  - Requires the dependencies be dynamically determined

Garnet / Amulet Default Algorithm

Human-Computer Interaction Institute



- Variables in the dependencies
  - Example: D = p^.left + A
  - Important innovation in Garnet we invented, now ubiquitous
  - Supports feedback objects
    - outlineRect.left = selectedObject^.left …

      circle1.object_over = rect34
      circle1.left = self.object_over.right + 10
  - Supports loops: D = Max(components^)
  - Only evaluates needed part of conditionals
    width = if otherpart.value > tolerance
      then *expensive computation*
      else otherpart.width
  - Requires the dependencies be dynamically determined

Examples of Expressing Constraints

- Garnet:

```
(create-instance NIL opal:line
        (:points '(340 318 365 358))
        (:grow-p T)
        (:x1 (o-formula (first (gvl :points))))
        (:y1 (o-formula (second (gvl :points))))
        (:x2 (o-formula (third (gvl :points))))
        (:y2 (o-formula (fourth (gvl :points)))))
```

- Amulet:

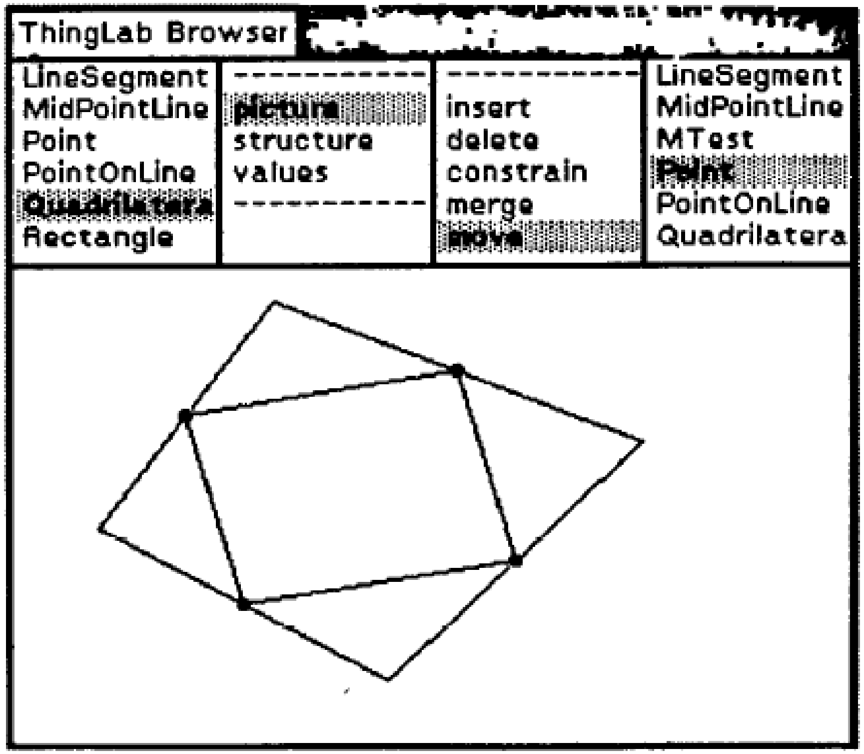
```
Am_Define_Formula (int, height_of_layout) {
  int h = (int)Am_Height_Of_Parts(self) + 2 *
((int)self.Get(Am_TOP_OFFSET));
  return h < 75 ? 75 : h;
}

am_empty_dialog = Am_Window.Create("empty_dialog_window")
    .Set (Am_LEFT_OFFSET, 5) // used in width_of_layout
    .Set (Am_TOP_OFFSET, 5) // used in height_of_layout
    .Set (Am_WIDTH, width_of_layout)
    .Set (Am_HEIGHT, height_of_layout)
        ...
```

Other One-Way Variations

- Multiple outputs
  - (D1, D2, ... Dm) = F(I1, I2, ... In)
- Side-effects in the formulas
  - useful for creating objects
  - when happen?
  - what if create new objects with new constraints
  - cycles cannot be detected
- Constant formula elimination
  - To decrease the size used by constraints

Two-Way (Multi-way) Constraints

- From ThingLab (~1979)
  - Alan Borning. “Defining Constraints Graphically,” *Human Factors in Computing Systems. Boston, MA, Apr, 1986. pp. 137-143. Proceedings SIGCHI'86.*
- Constraints are expressions with multiple variables
- Any may be modified to get the right values
- Example: A.right = A.left + A.width - 1
- Often requires programmer to provide methods for solving the constraint in each direction:
  A.left = A.right - A.width + 1
  A.width = A.right - A.left + 1
- Useful if mouse expressed as a constraint



Two-Way implementations

- Requires a *planning* step to decide which way to solve
  - Many systems compute plans and save them around since usually change same variable repeatedly
- In general, have a graph of dependencies, find a path through the graph
- How control which direction is solved? CurrentSliderVal = mouseX - scrollbar.left
  - "Constraint hierarchies" = priorities
    - constants, interaction use "stay" constraints with high priority
  - Dynamically add and remove constraints
- Brad Vander Zanden's "QuickPlan" solver
  - Handles multi-output, multi-way cyclic constraints in $O(n^2)$ time instead of exponential like previous algorithms

Simultaneous Equations

- Required for parallel, perpendicular lines; tangency, etc.
- Also for aggregate's size
- Numerical (relaxation) or symbolic techniques
  - Thinglab bridge (1979) (cite)

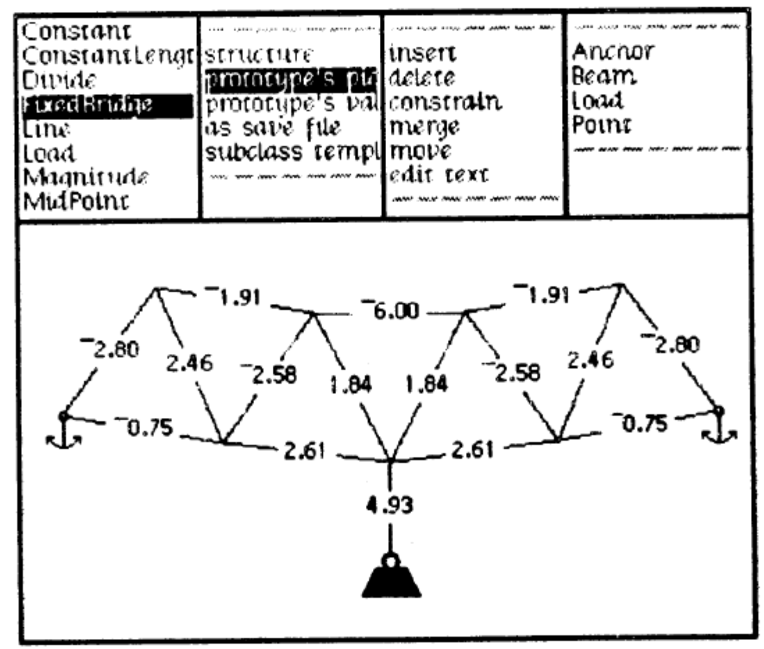
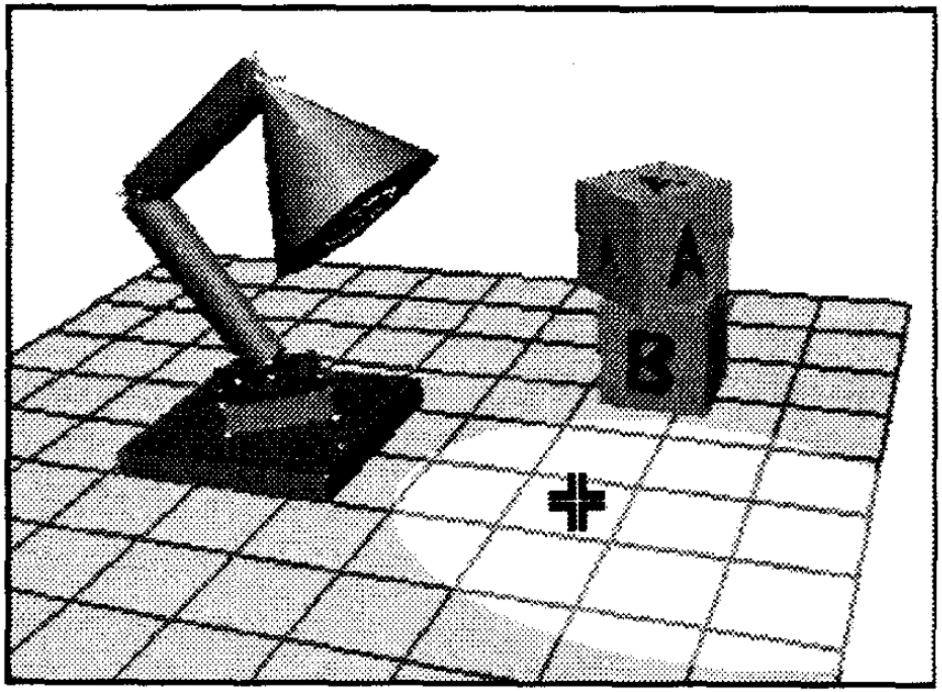


Figure 2.31 - A bridge under load

Incremental

- Michael Gleicher's PhD thesis, 1994
- Only express forward computations
- Tries to get reverse by incrementally changing the forward computation in the right direction using derivatives.
- Supports interactions otherwise not possible
- Produces smooth animations



Animation Constraints in Amulet

- Implemented using Amulet's constraint mechanism
- When slot set with a new value, restores old value, and animates from old to new value
- Usually, linear interpolation
- For colors, through either HSV or RGB space
- For visibility, various special effects between TRUE and FALSE
- *Demo*

Other Forms 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

Implementation Note

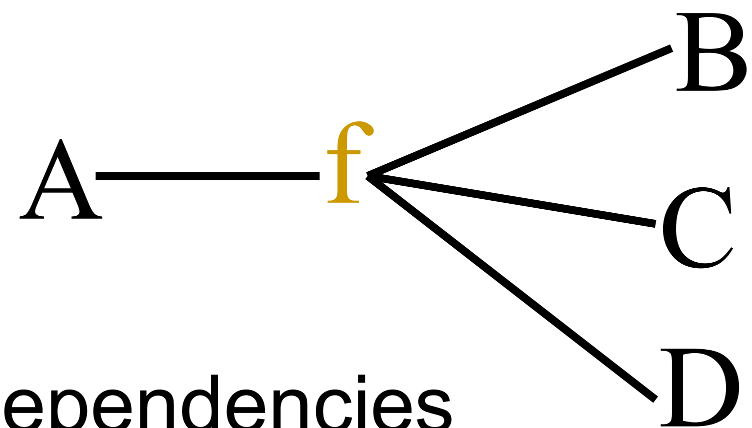
- Implementation details (the rest of these slides) will *not* be on the final test

Dependency graphs for Implementation

- 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)

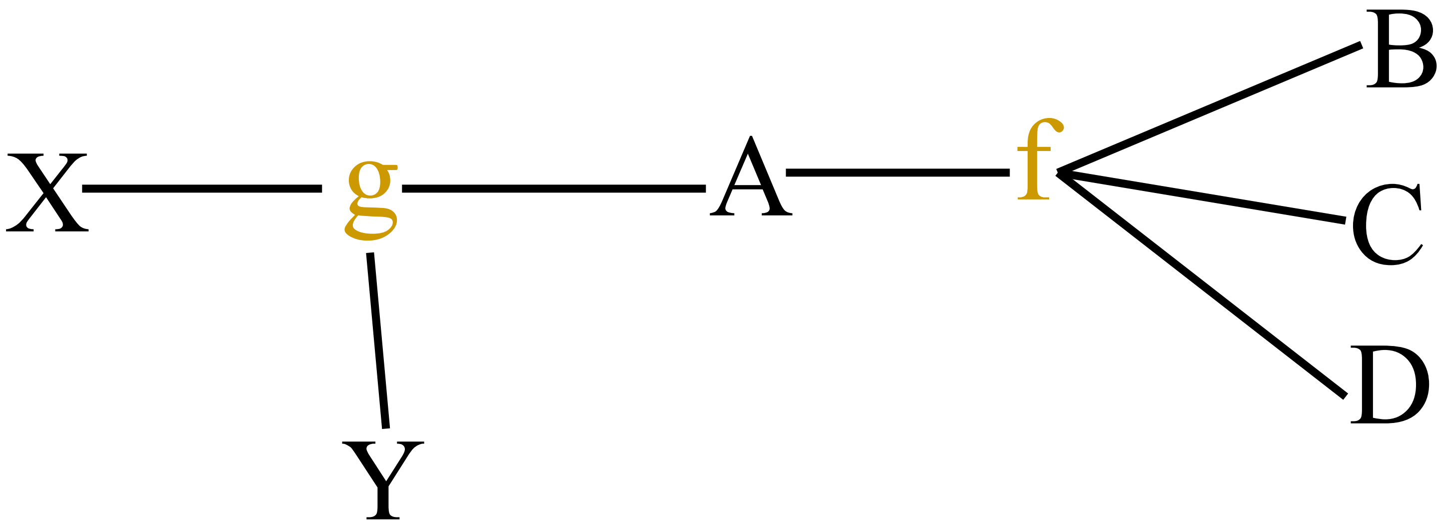


Edges are dependencies

Dependency graphs

Dependency graphs chain together:

$$X = g(A, Y)$$

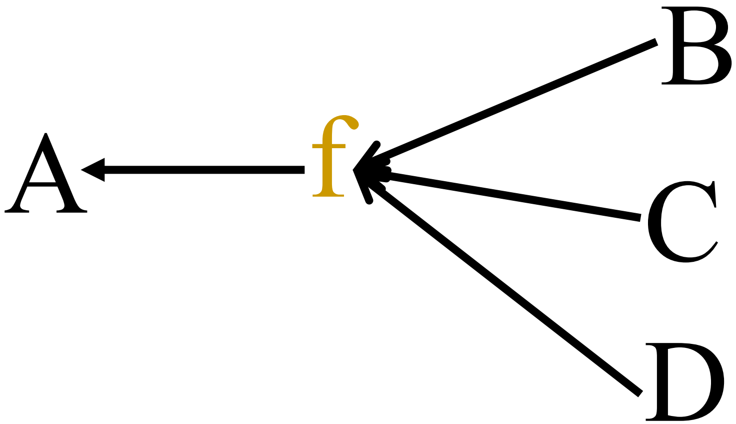


Kinds of constraint systems

- 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

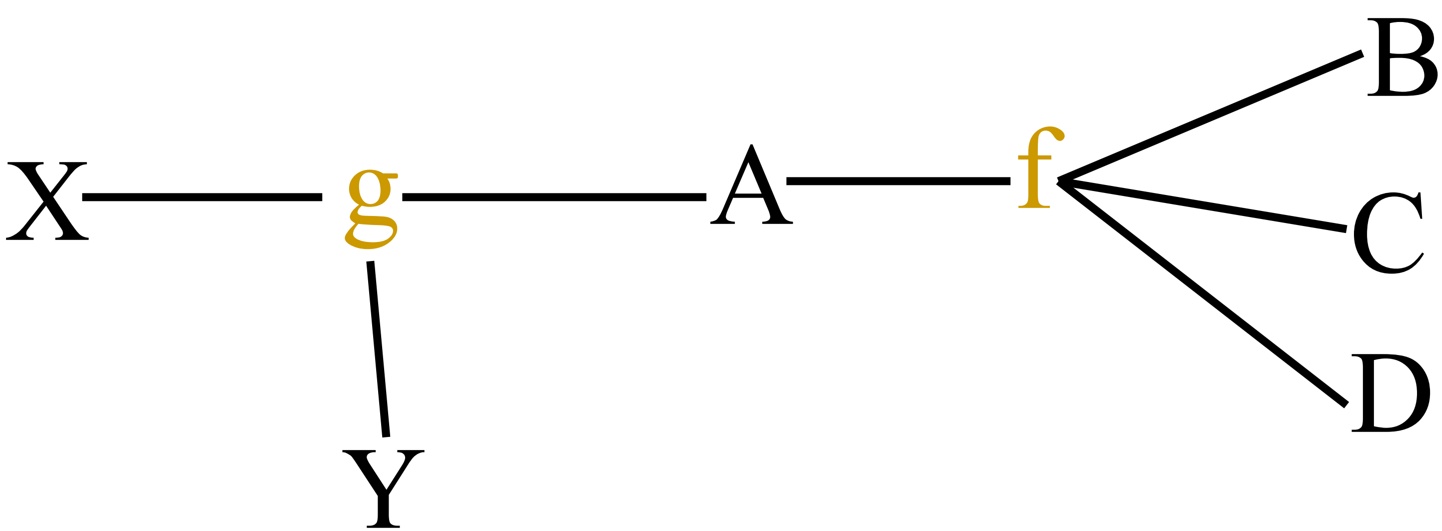
Multi-way constraints

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

Multi-way constraints

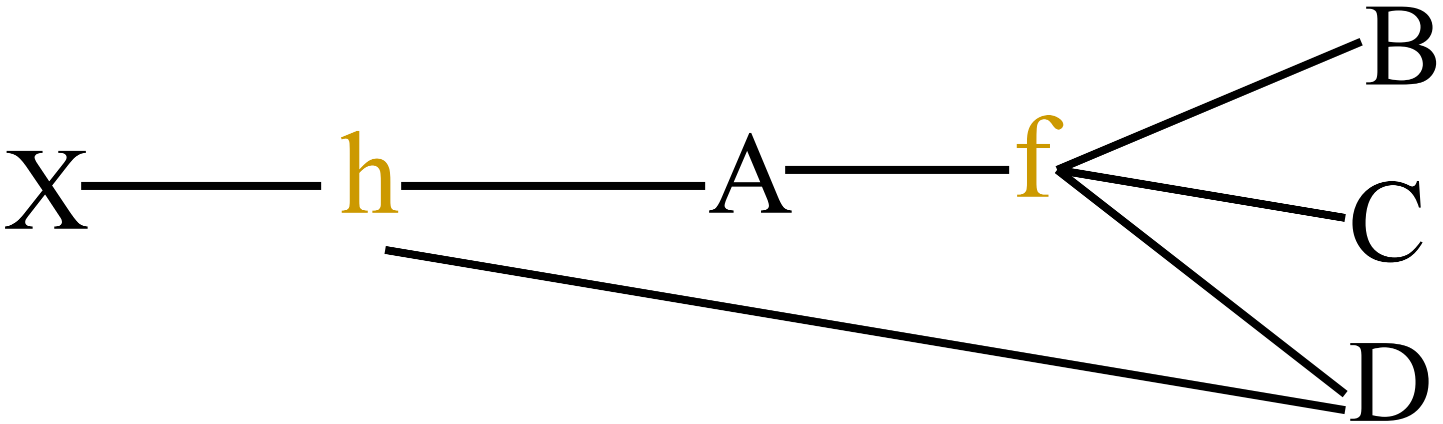
But all is not rosy

- 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

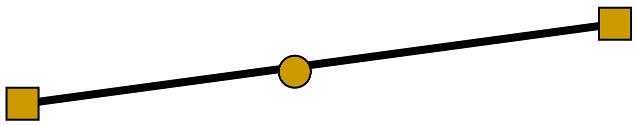
- A set of constraints can be:
  - Over-constrained
    - No valid solution that meets all constraints
  - Under-constrained
    - More than one solution
      - sometimes infinite numbers

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

Incoming Edges

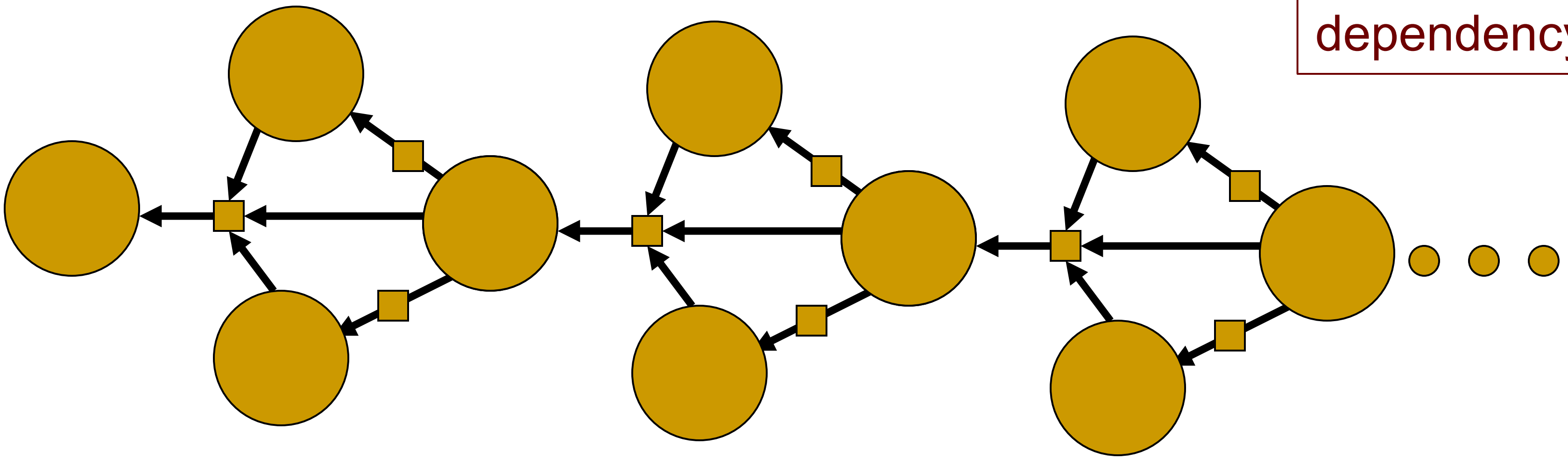
Naïve algorithm

```
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
```

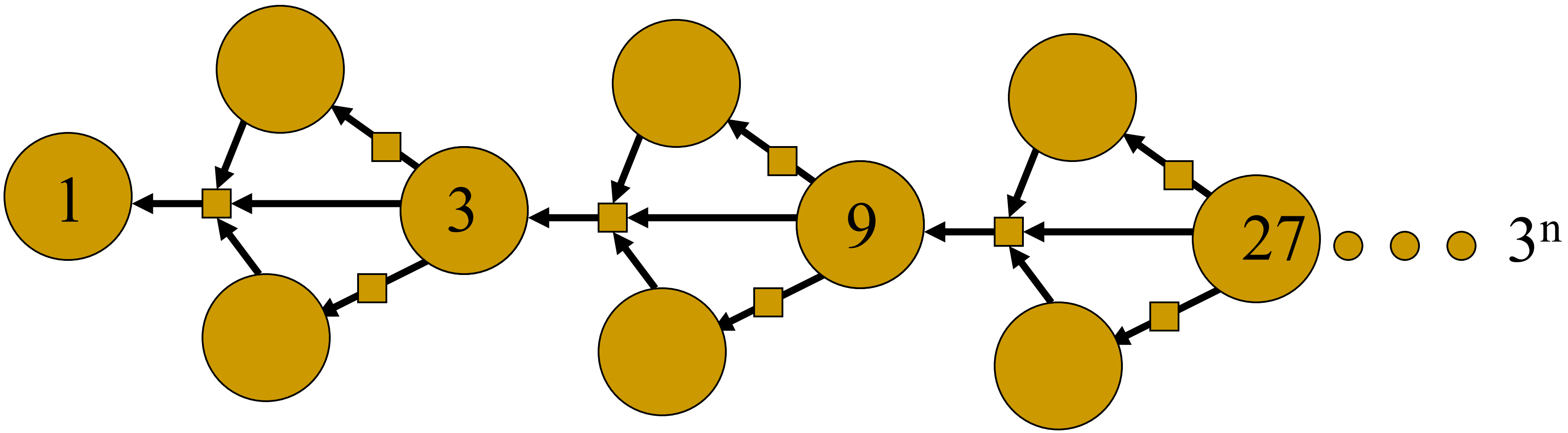
Why is this not a good plan?

Exponential Wasted Work

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

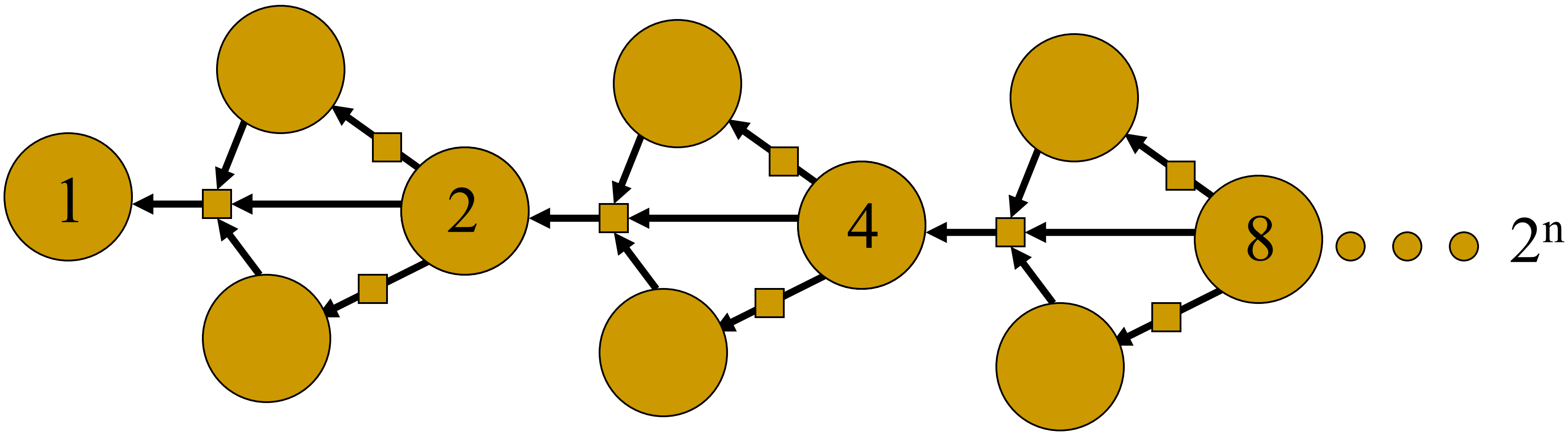


Exponential Wasted Work



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

Outgoing Edges

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

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)

Incoming Edges

```
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)

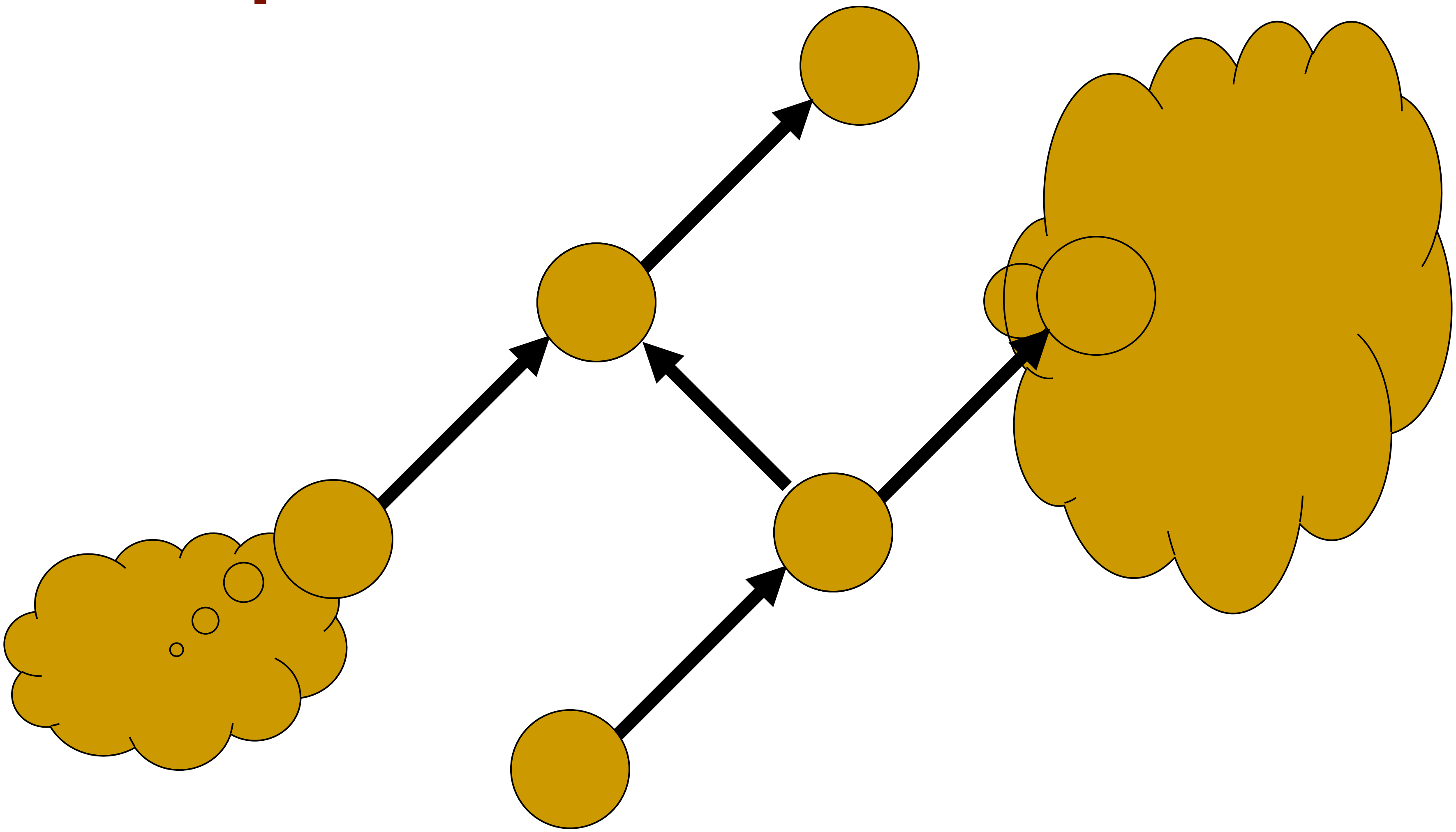
```
UpdateIfPending(v,Parms):
  pendingIn = false  //any incoming pending?
  For each incoming dep edge E do
    pendingIn |= E.pending
    E.pending = false

  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)

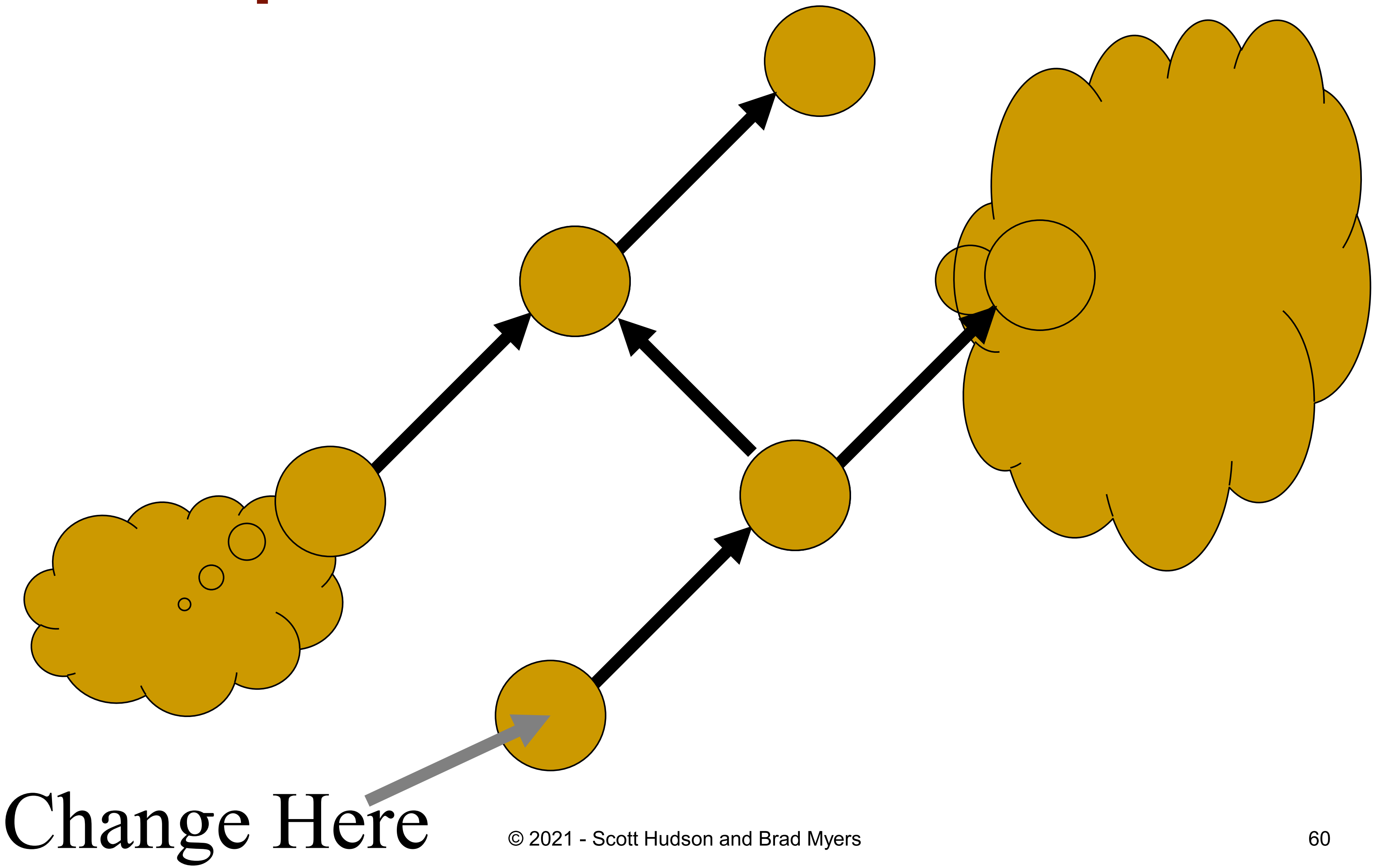
```
UpdateIfPending(v,Parms):
  pendingIn = false  //any incoming pending?
  For each incoming dep edge E do
    pendingIn |= E.pending
    E.pending = false

  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

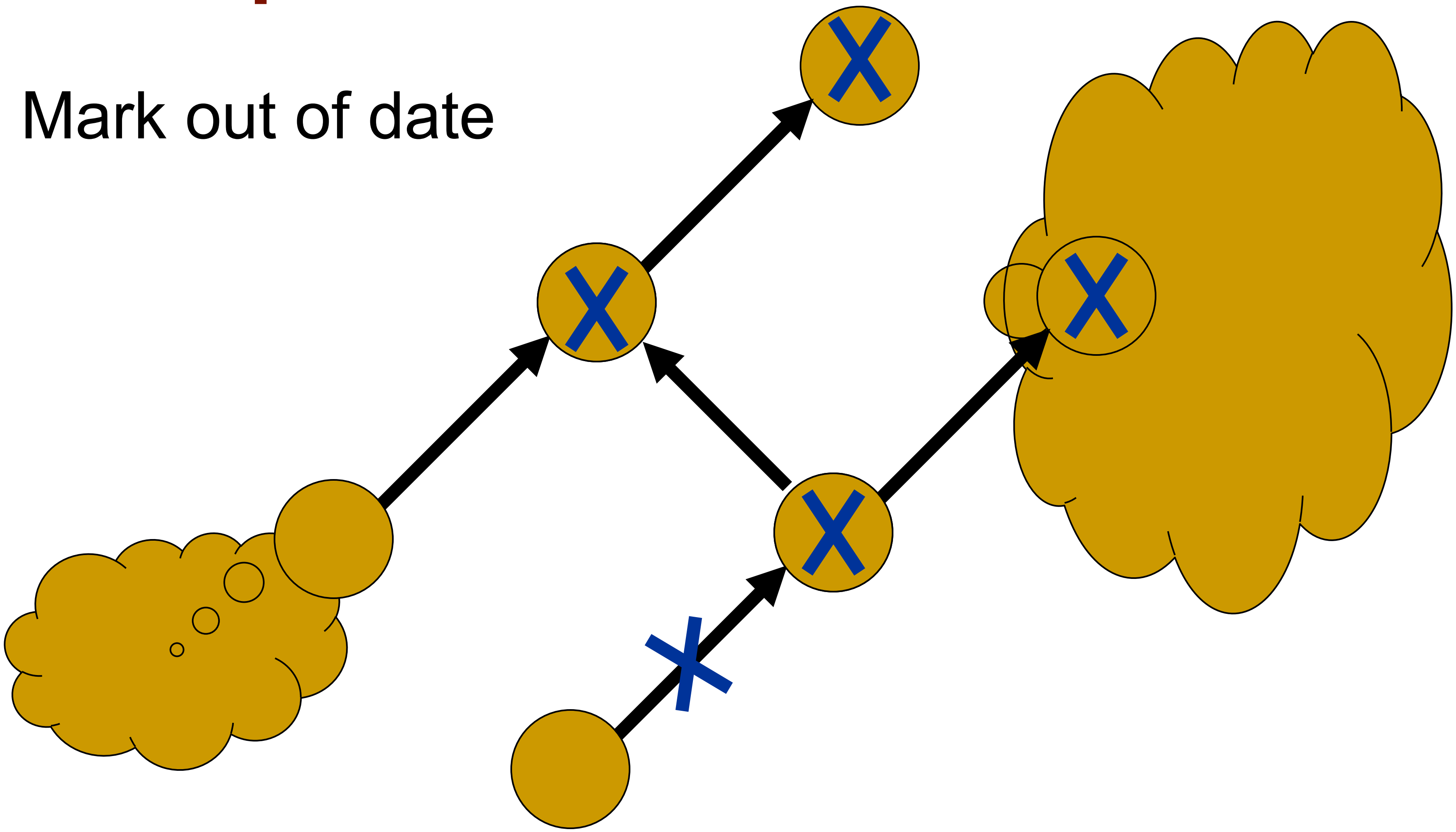


Example



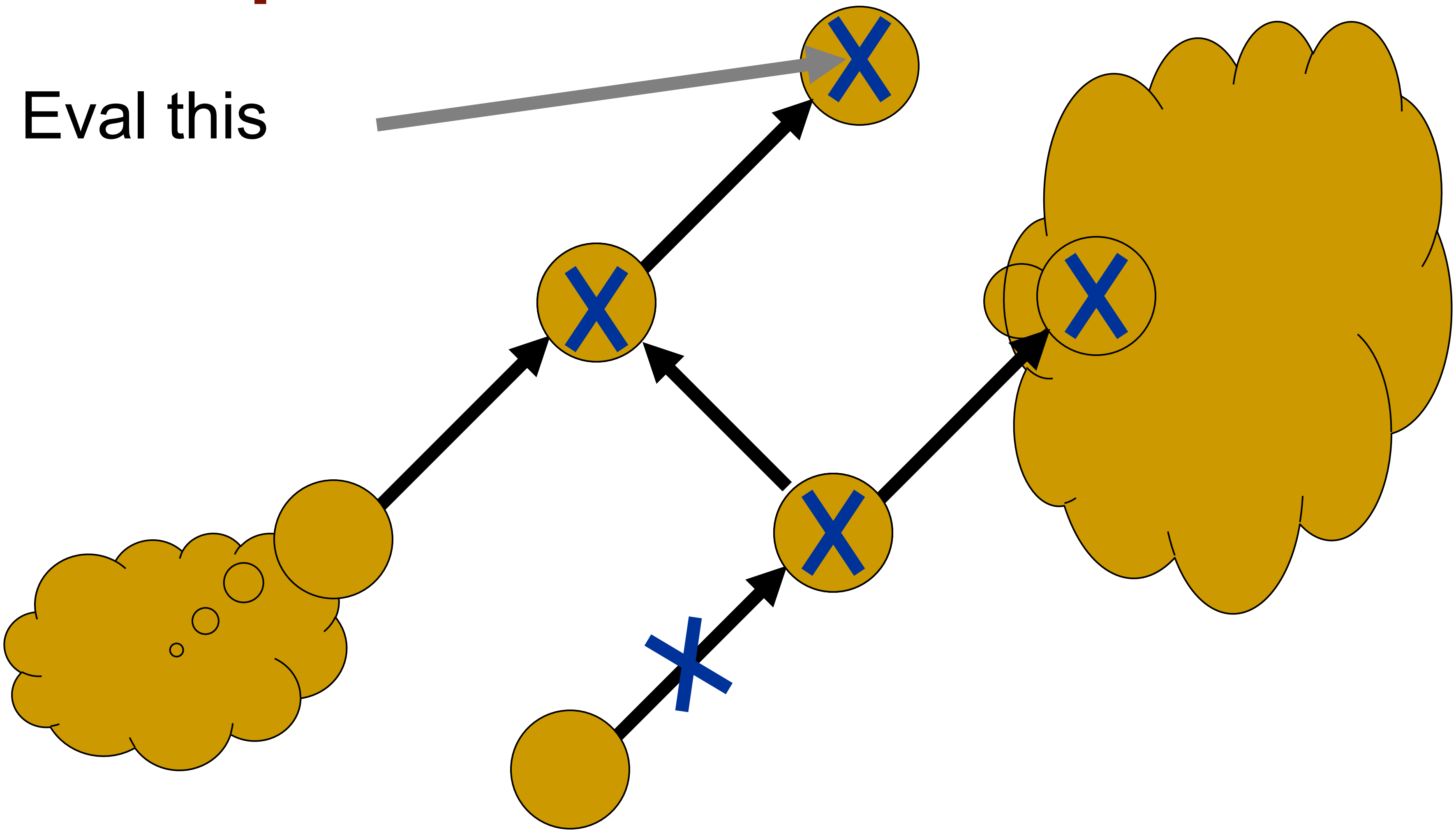
Example

Mark out of date

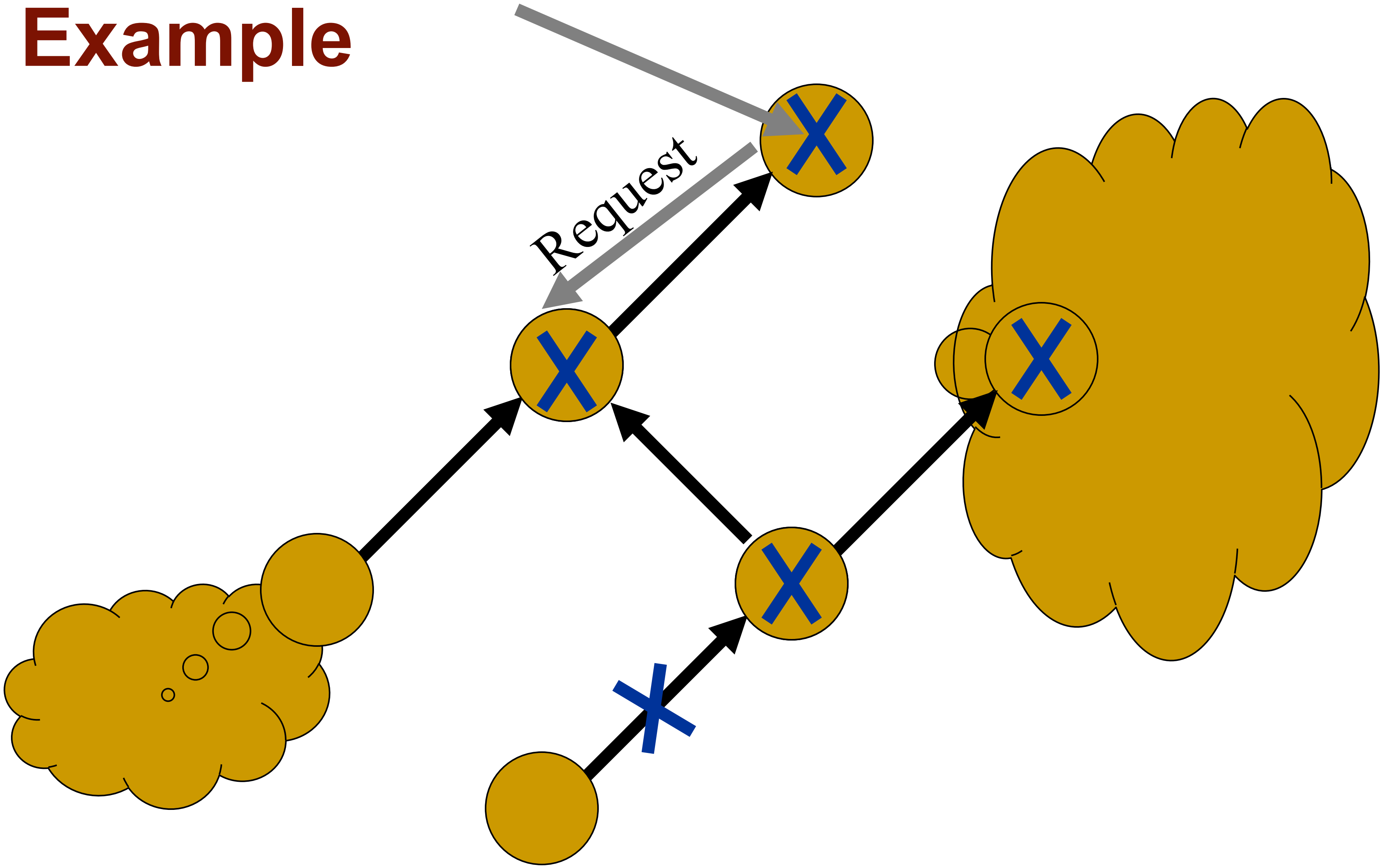


Example

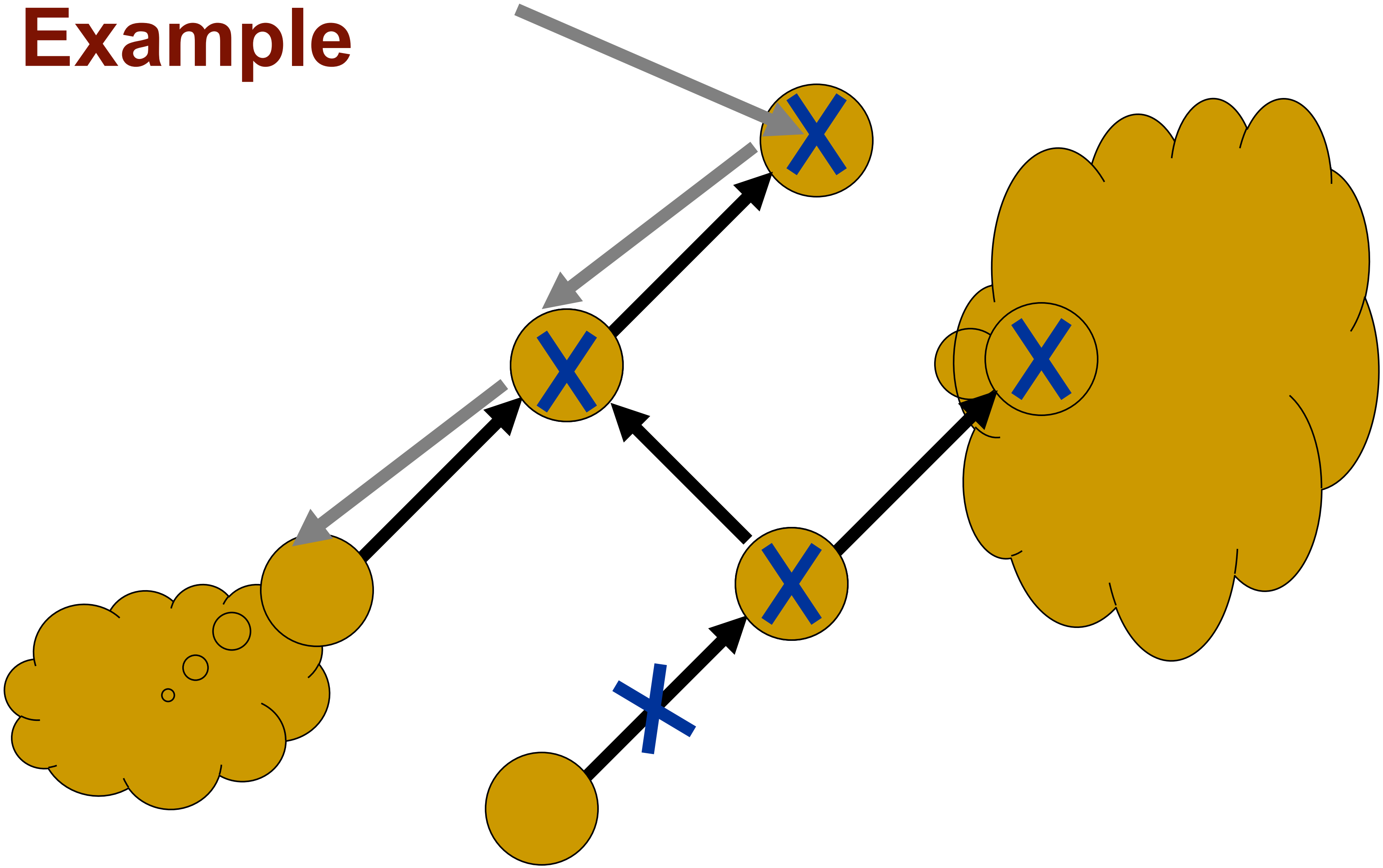
Eval this



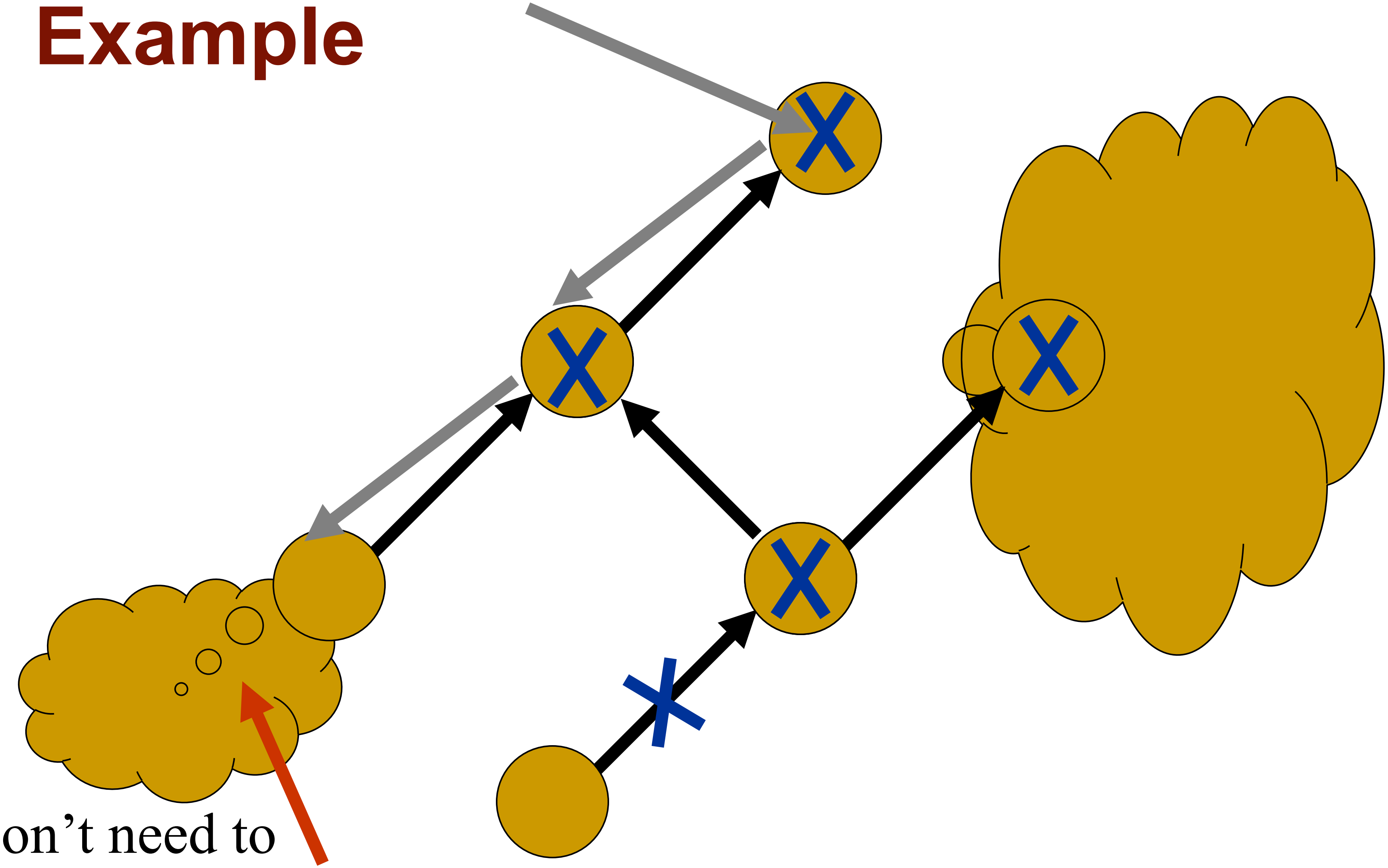
Example



Example

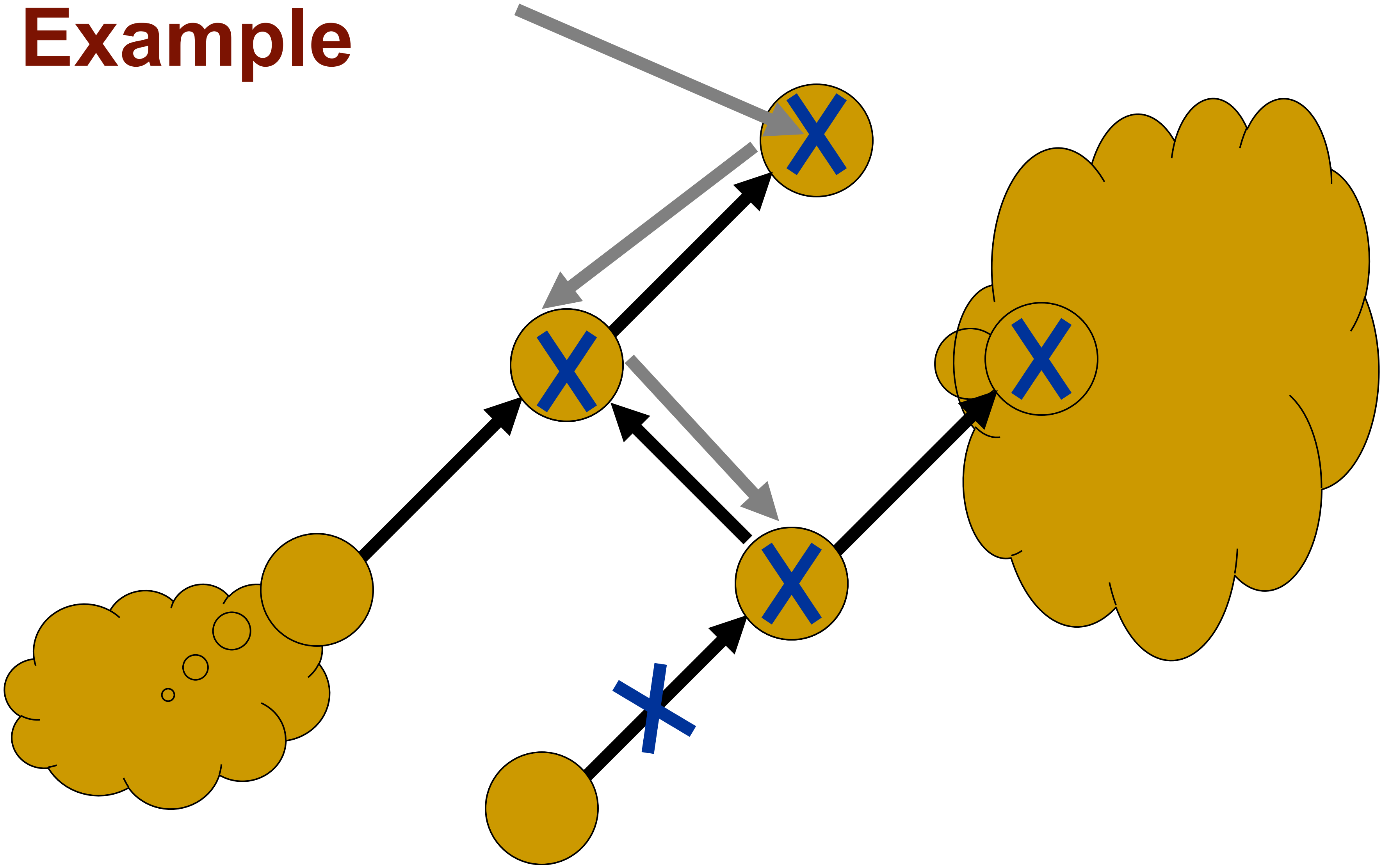


Example

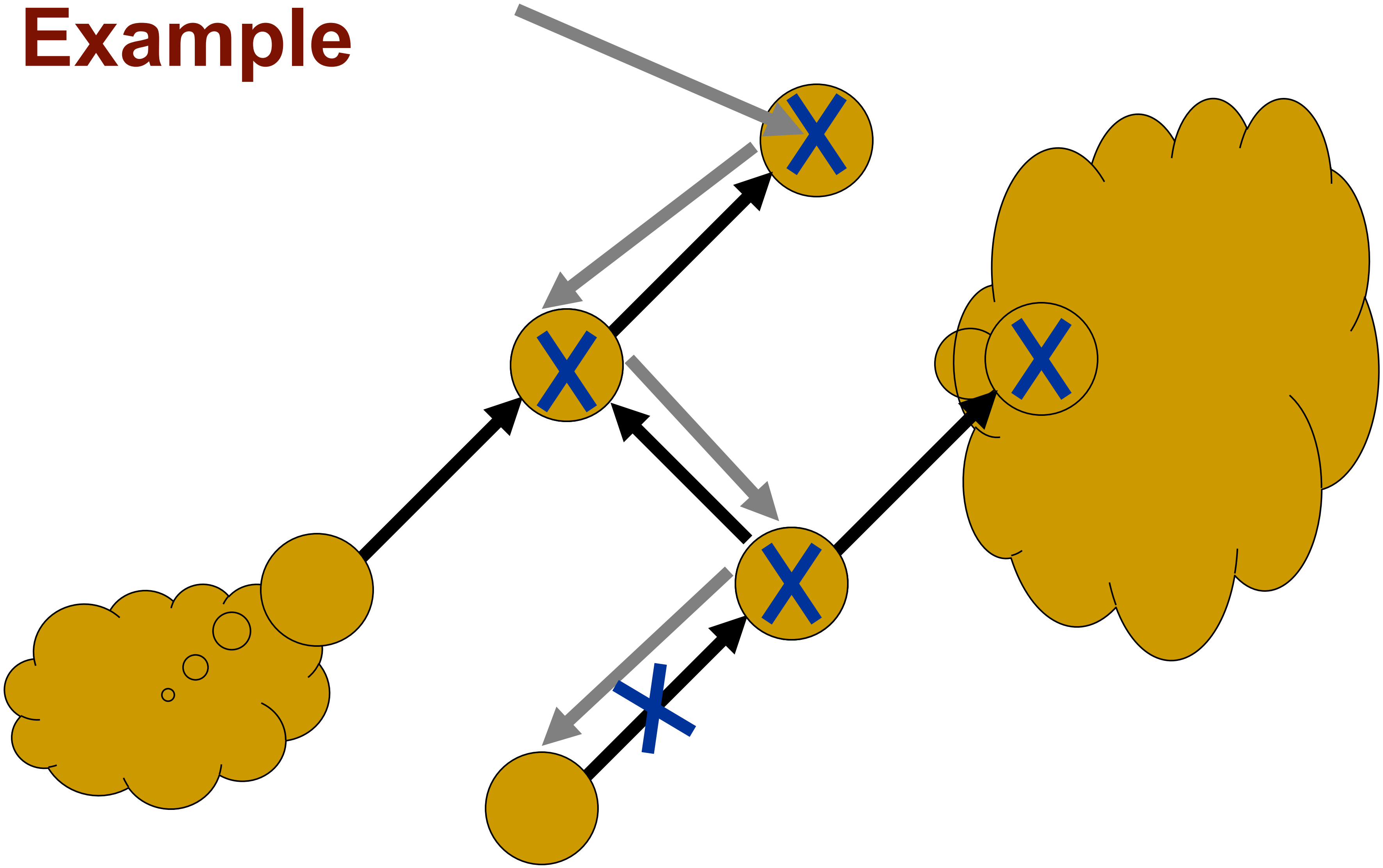


Don’t need to
eval any of these! (Not out-of-date)

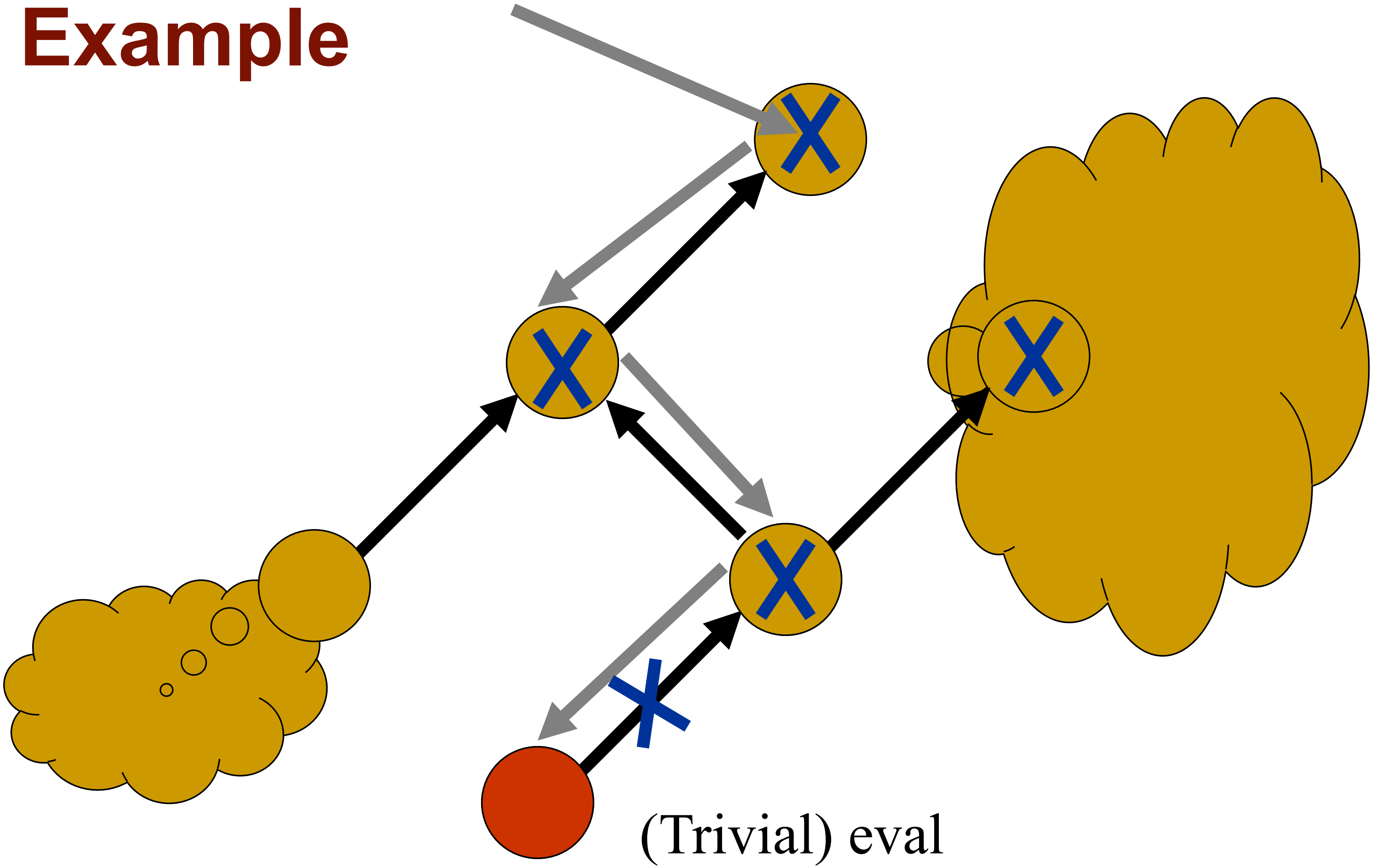
Example



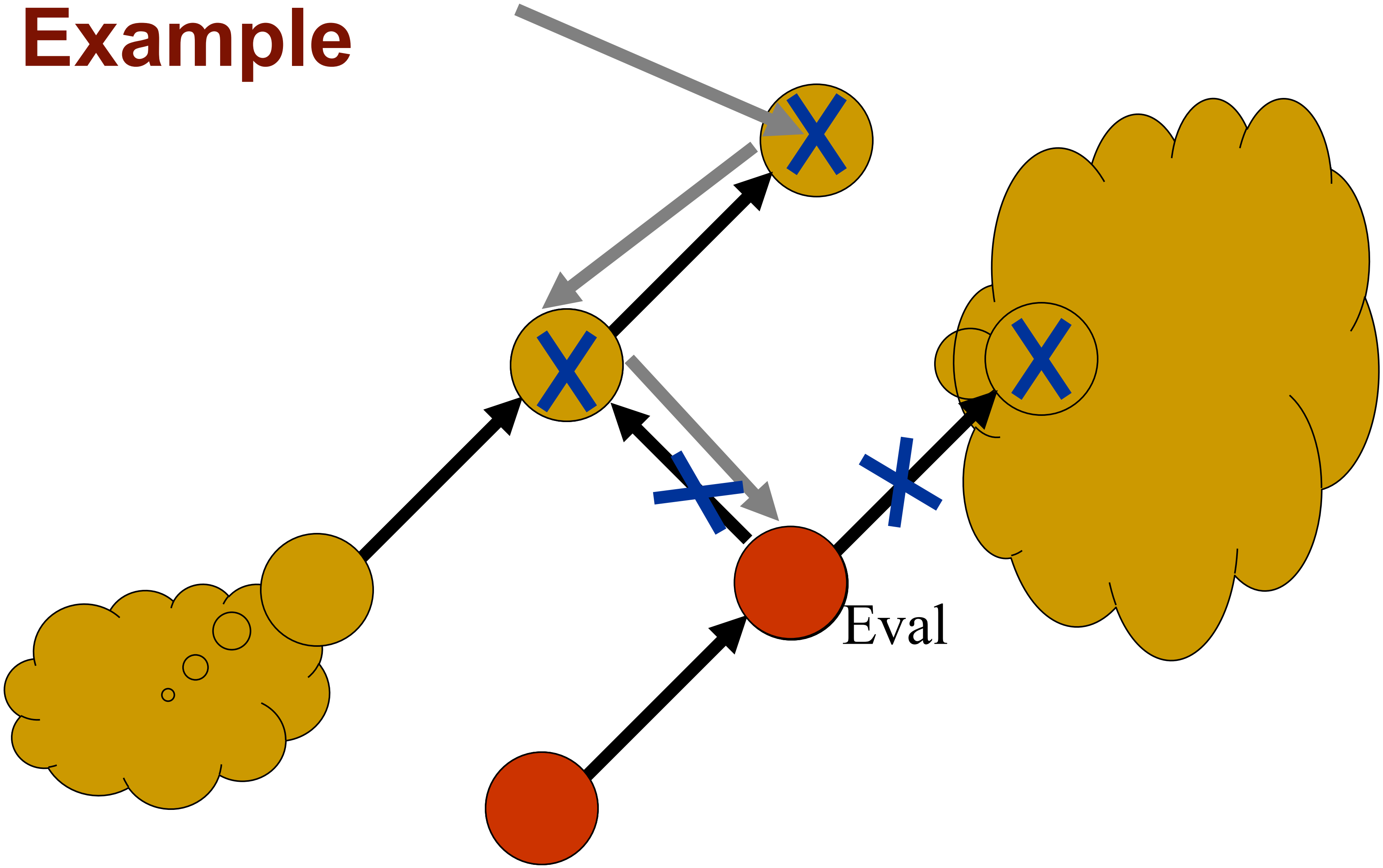
Example



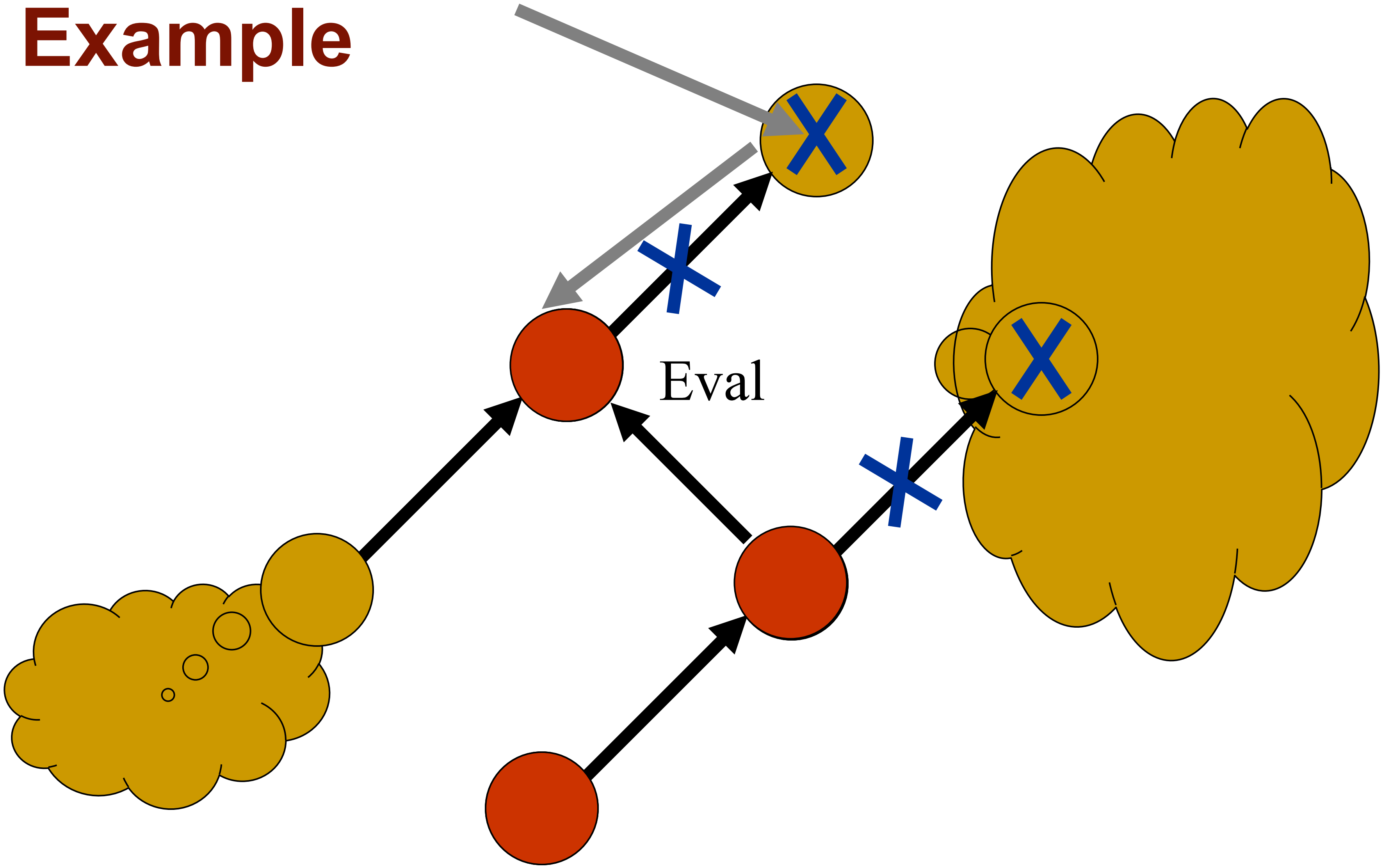
Example



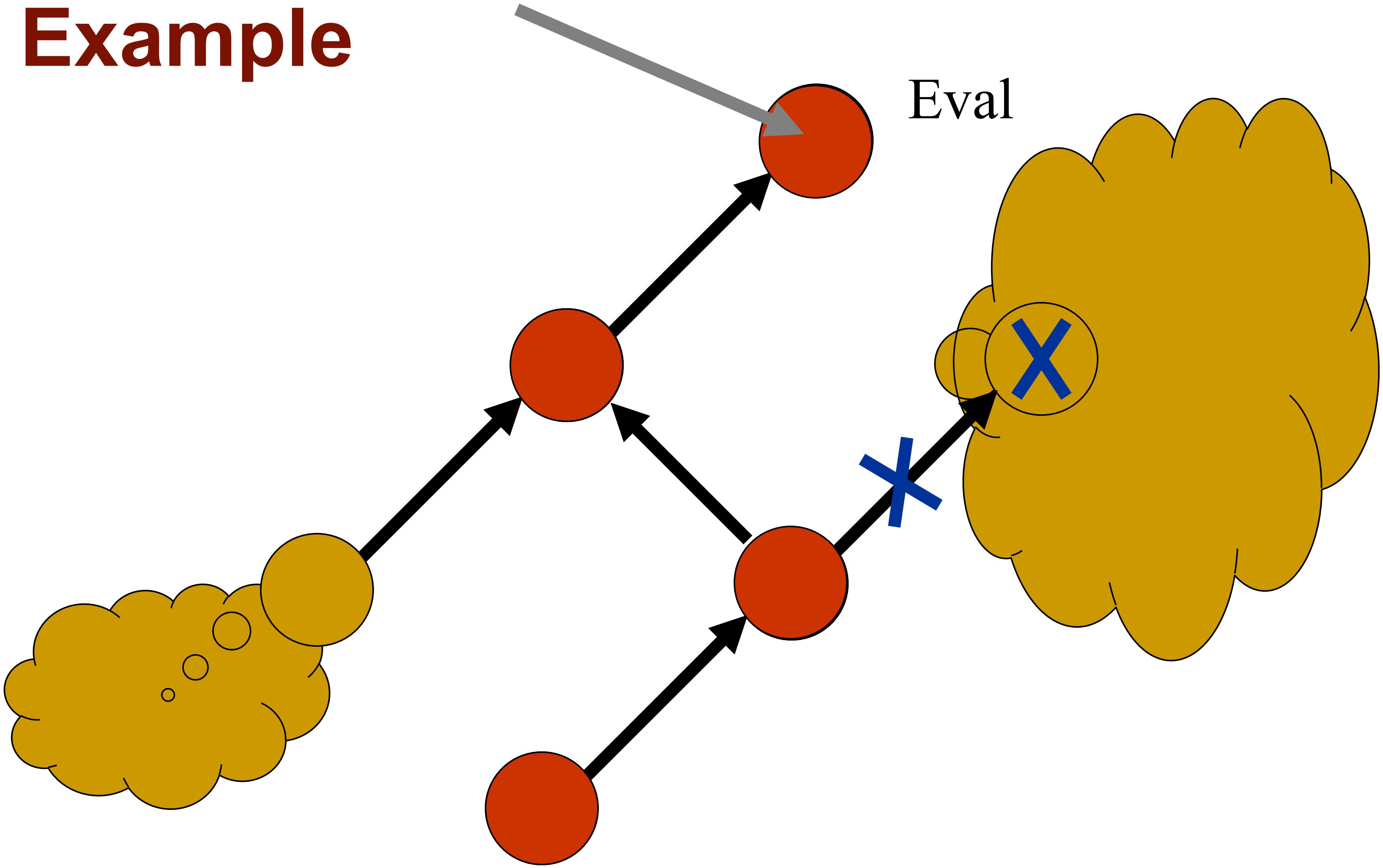
Example



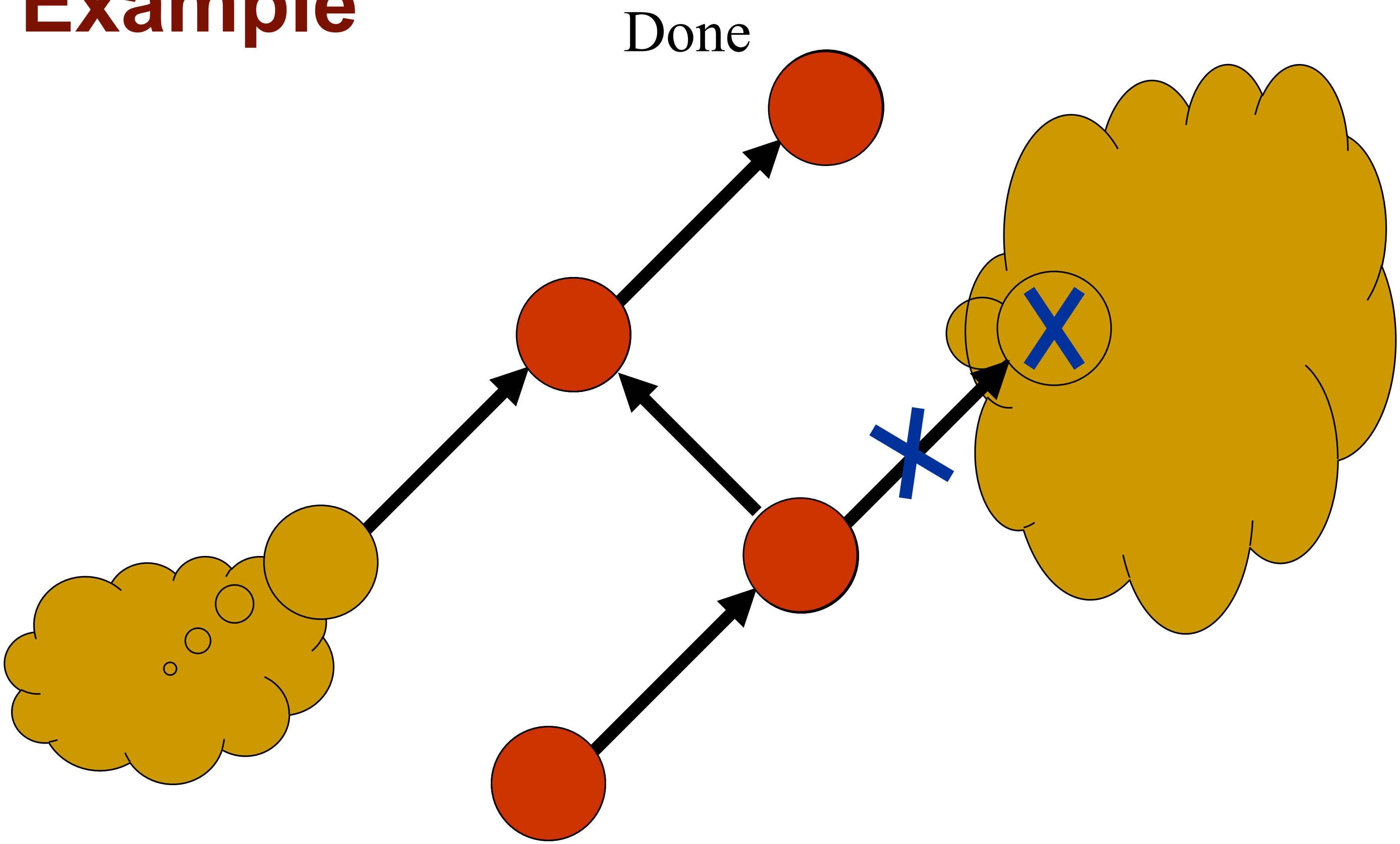
Example



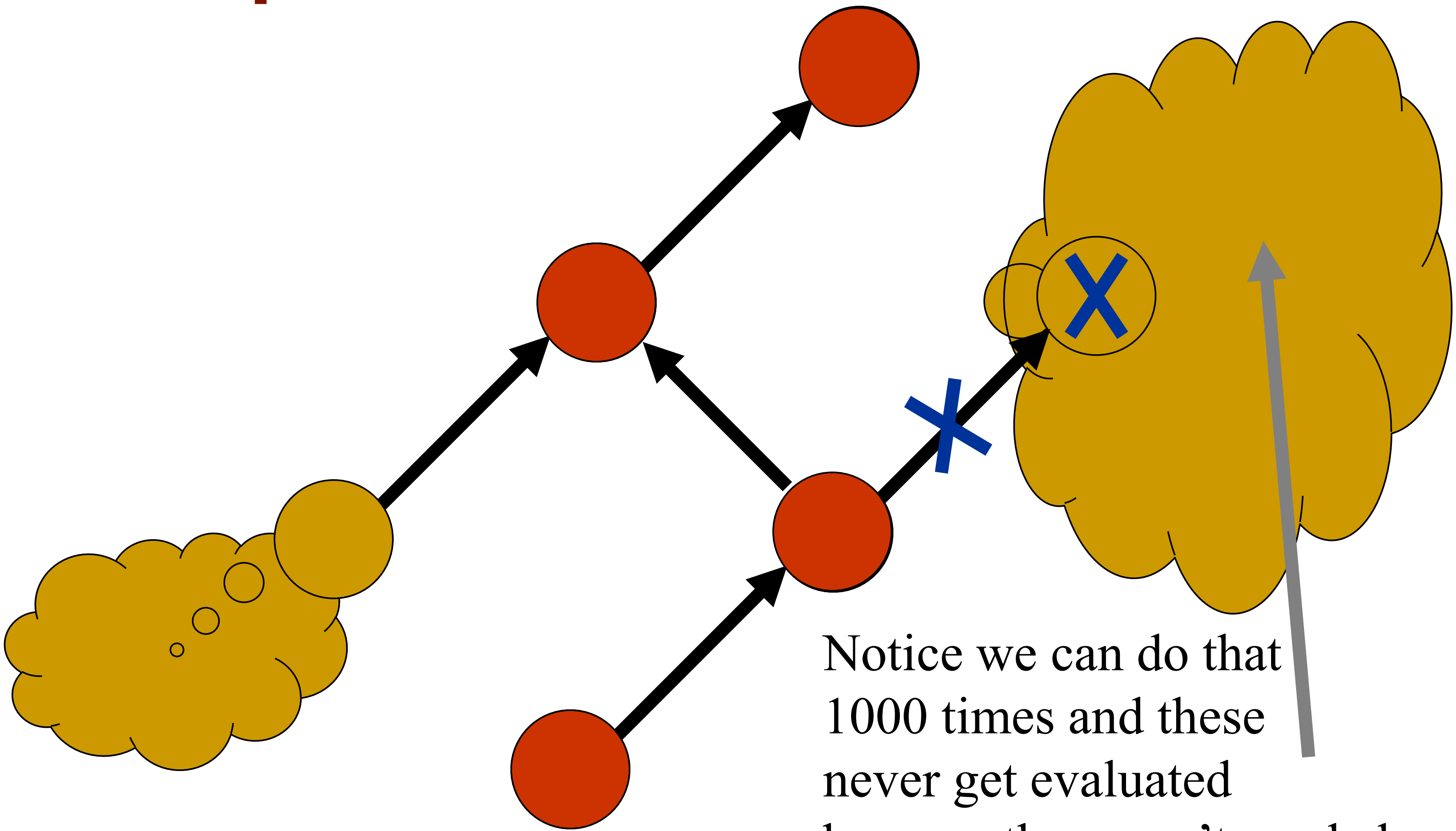
Example



Example

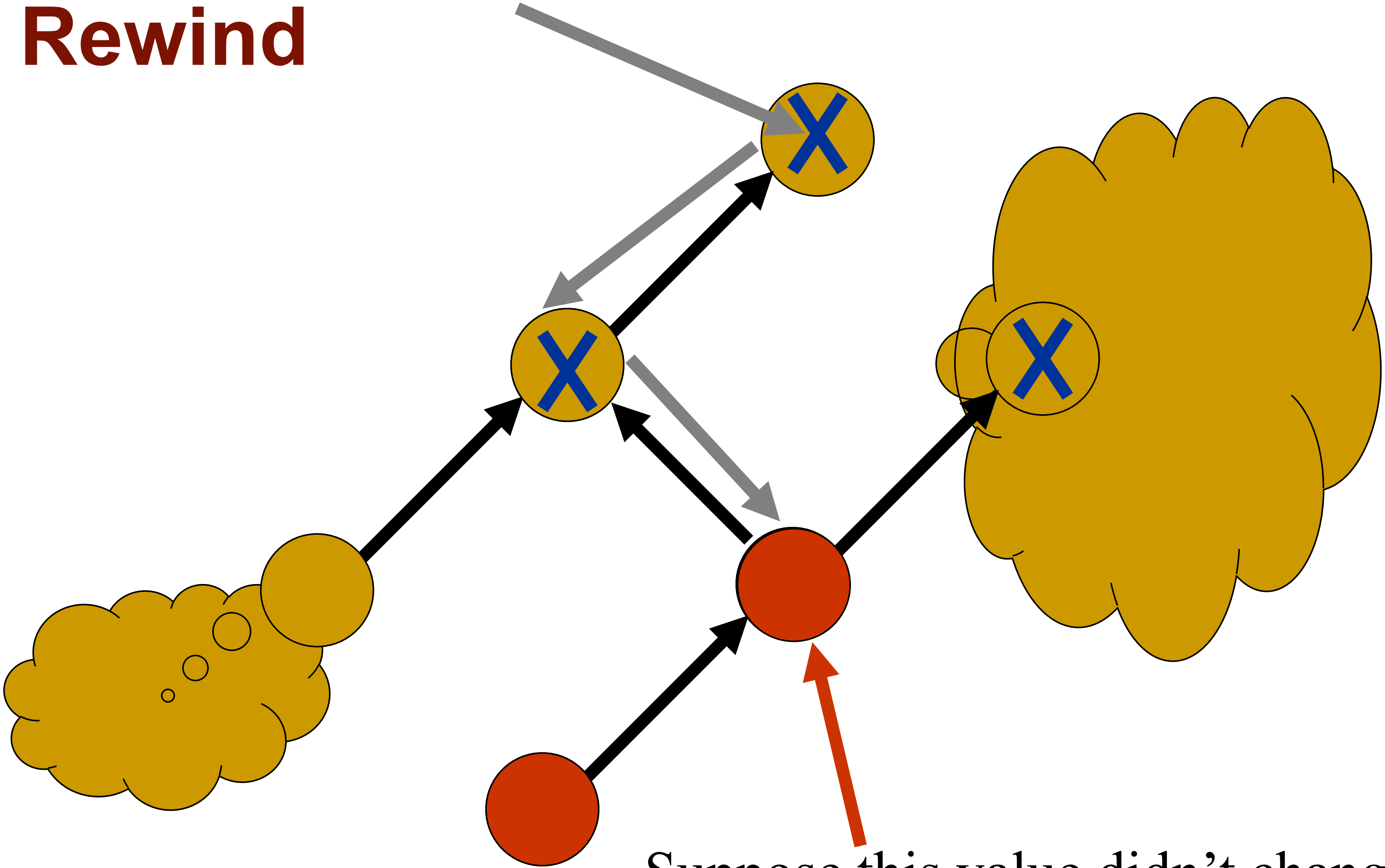


Example



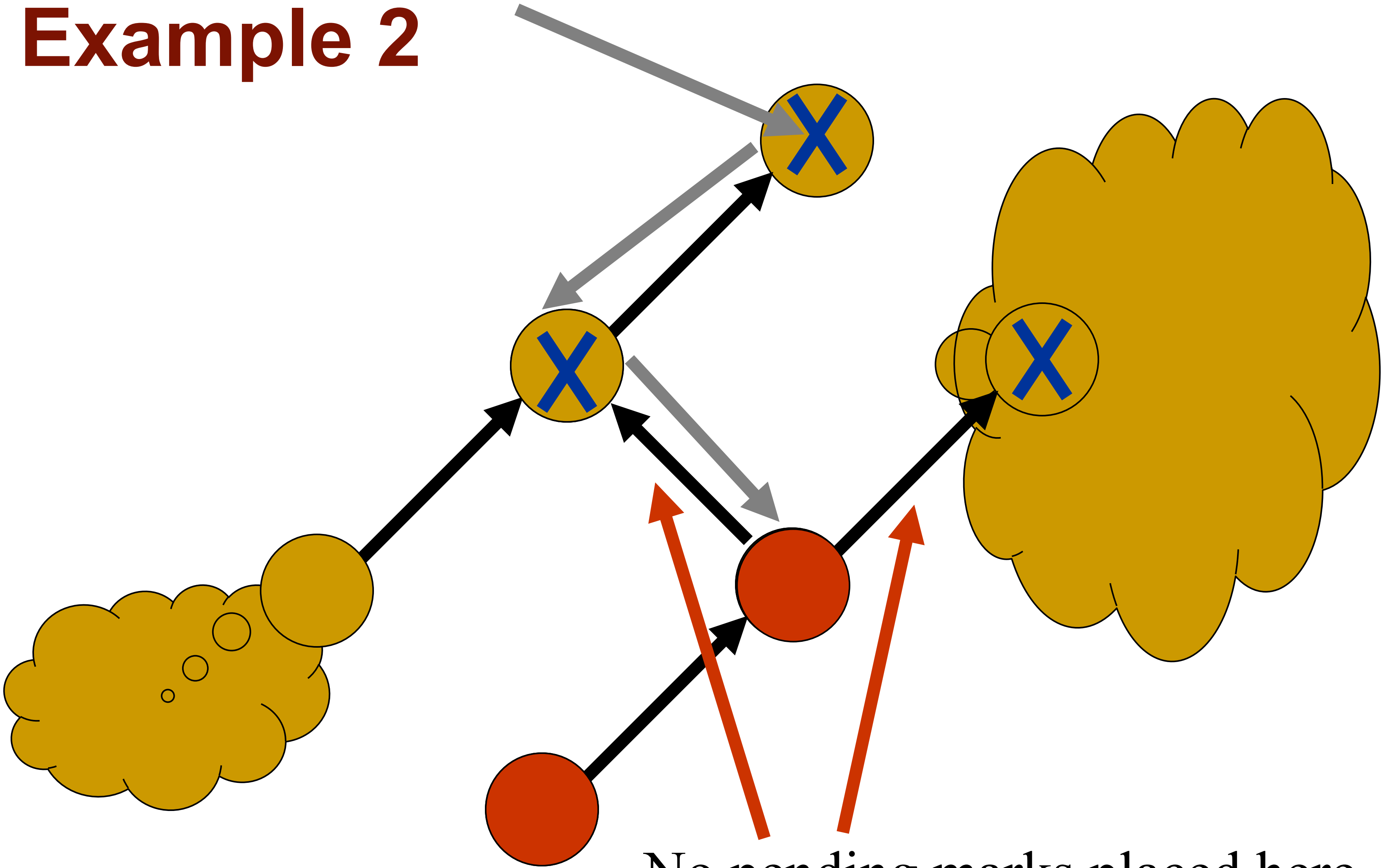
Notice we can do that 1000 times and these never get evaluated because they aren't needed

Rewind



Suppose this value didn't change

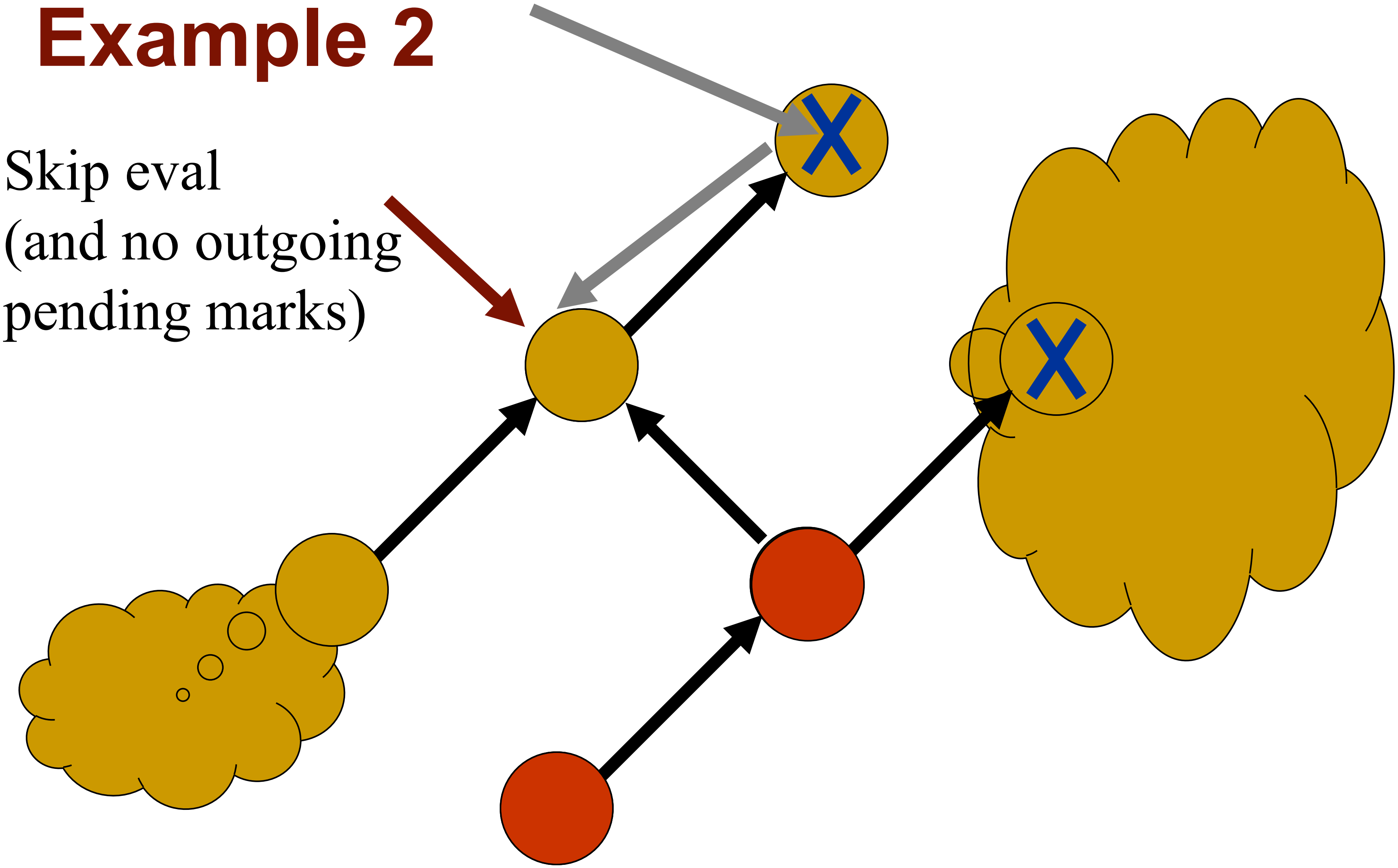
Example 2



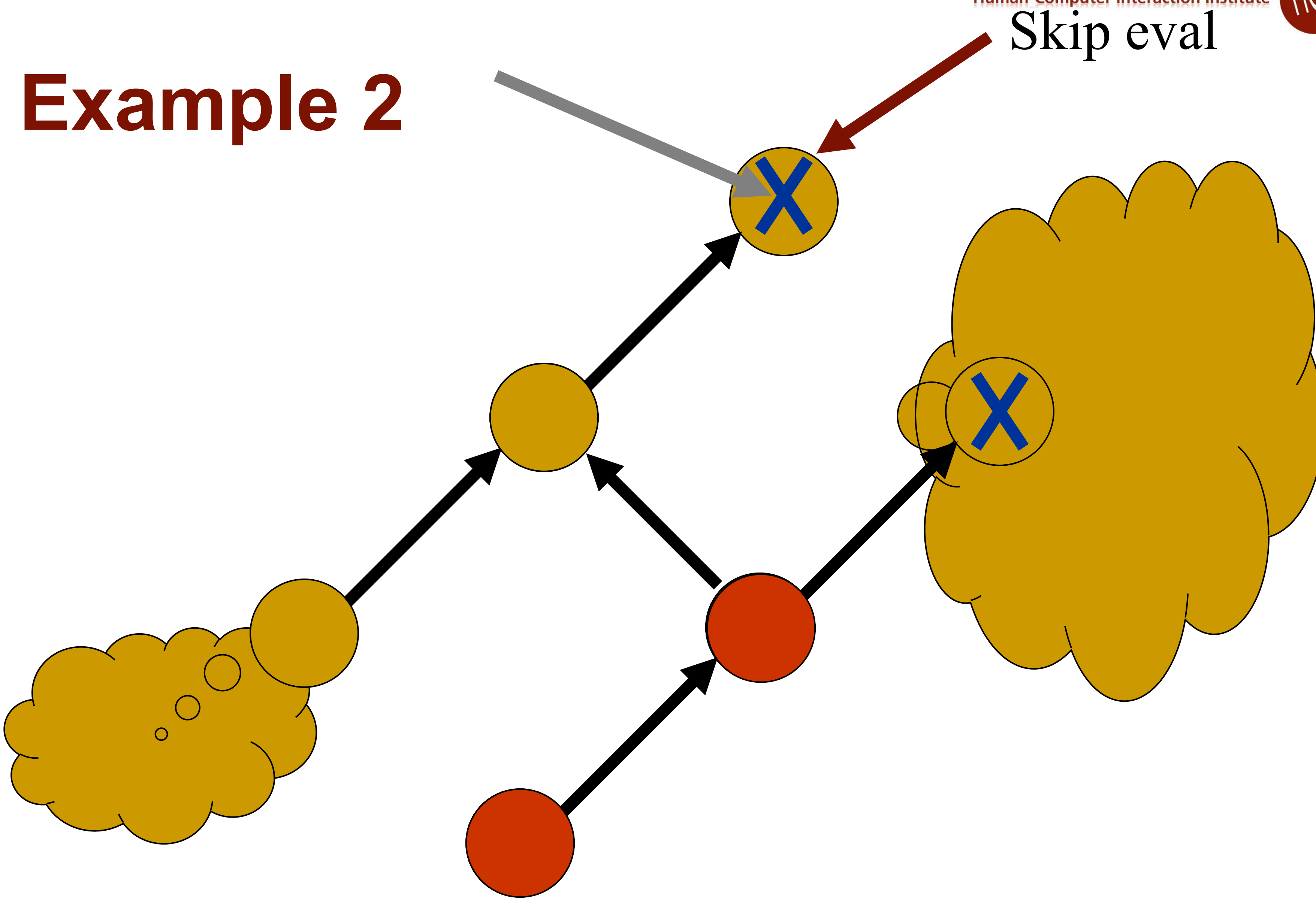
No pending marks placed here

Example 2

Skip eval
(and no outgoing
pending marks)



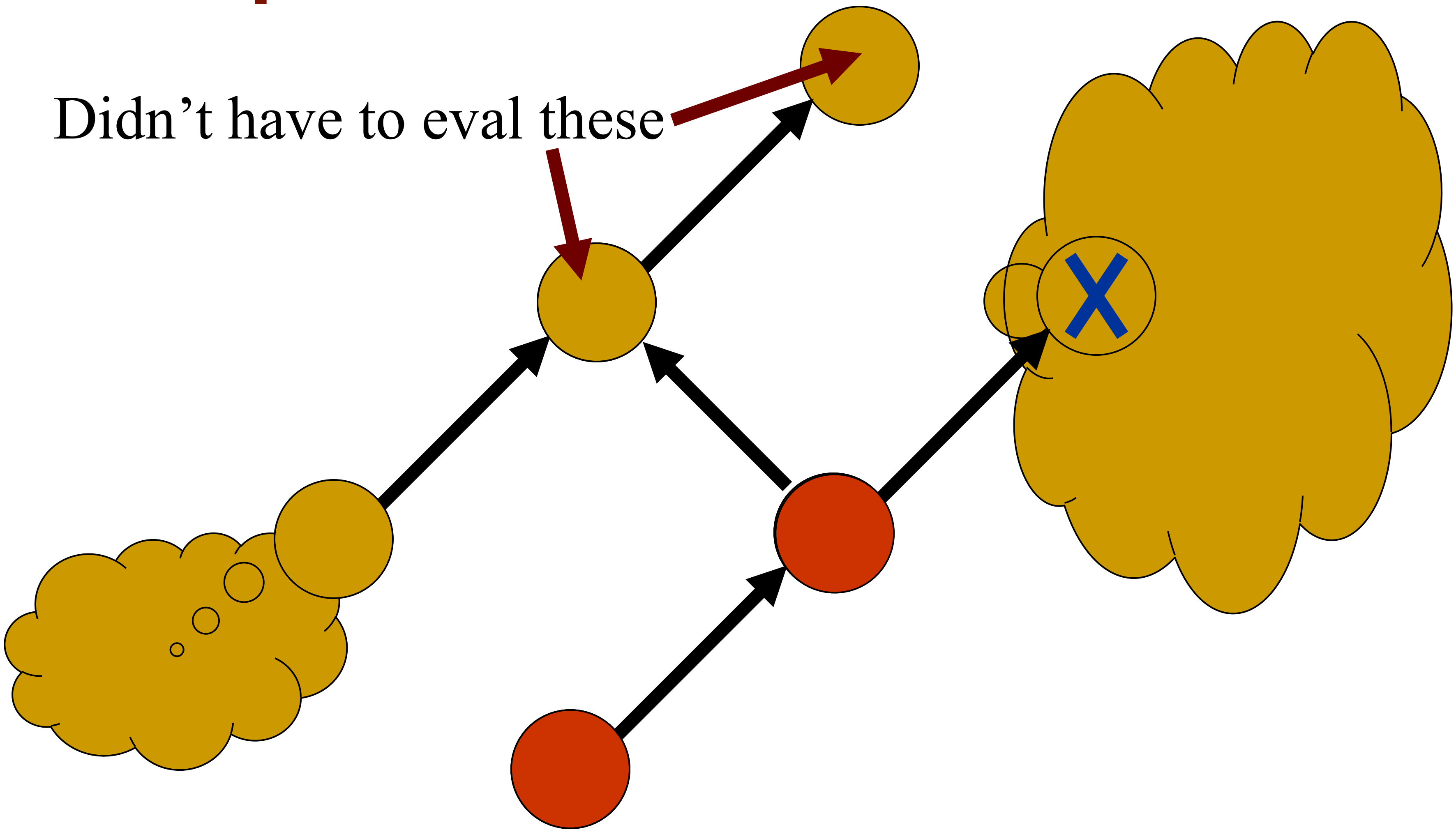
Example 2



Example 2

Done

Didn't have to eval these



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