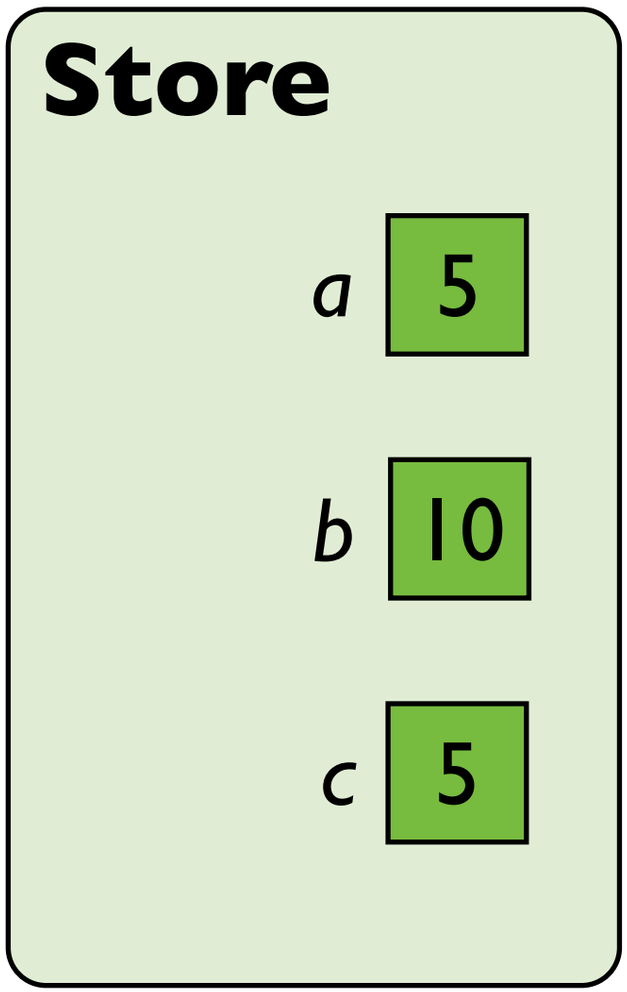
Software Verification (Fall 2013)
Lecture 7: Graph-Based Reasoning and Verification

Chris Poskitt

Chair of Software Engineering

ETH zürich

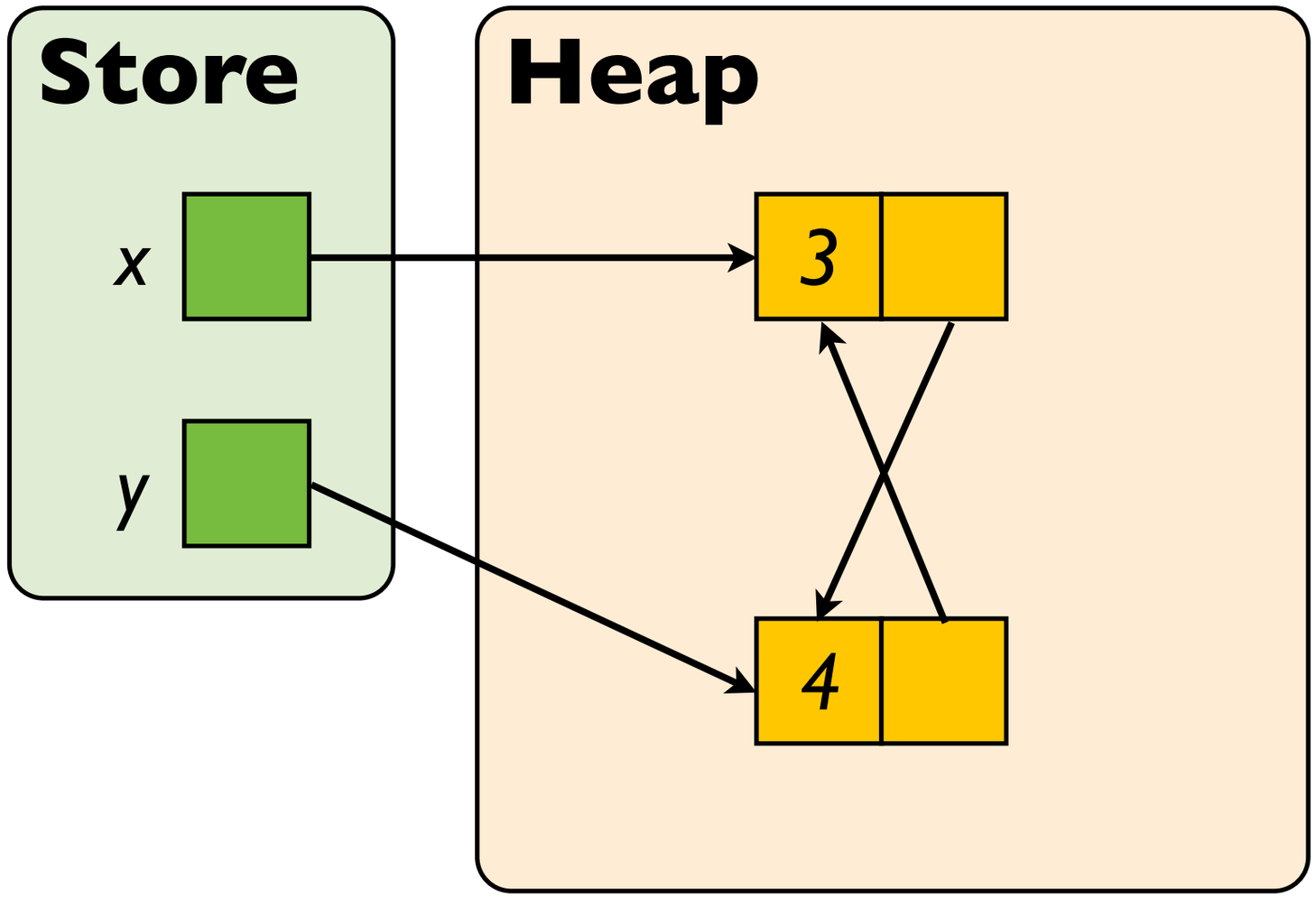
Programs we've reasoned about so far: (1) store-manipulating programs



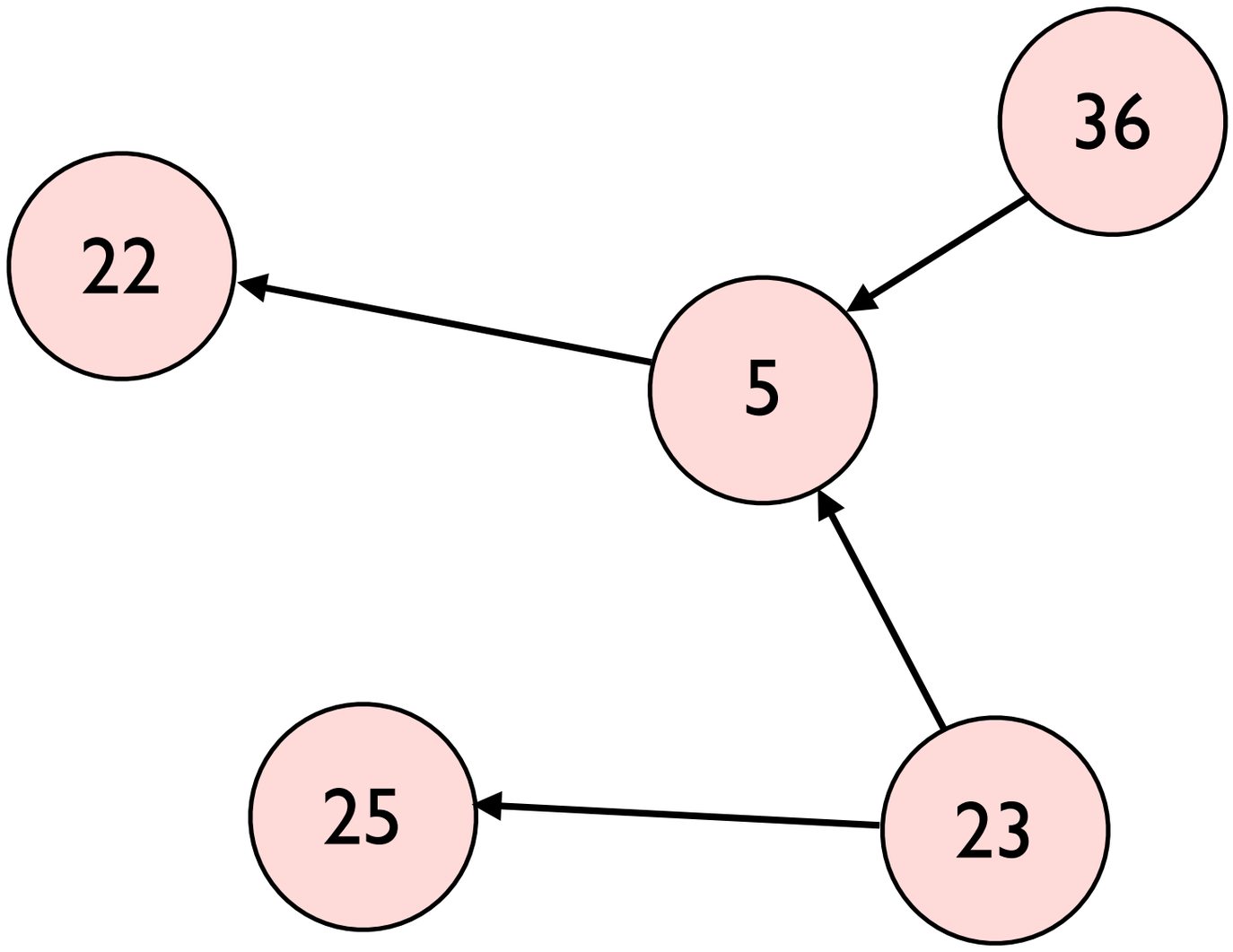

```
a := 5;
b := a*2;
c := a;
```

Programs we've reasoned about so far: (2) store+heap-manipulating programs

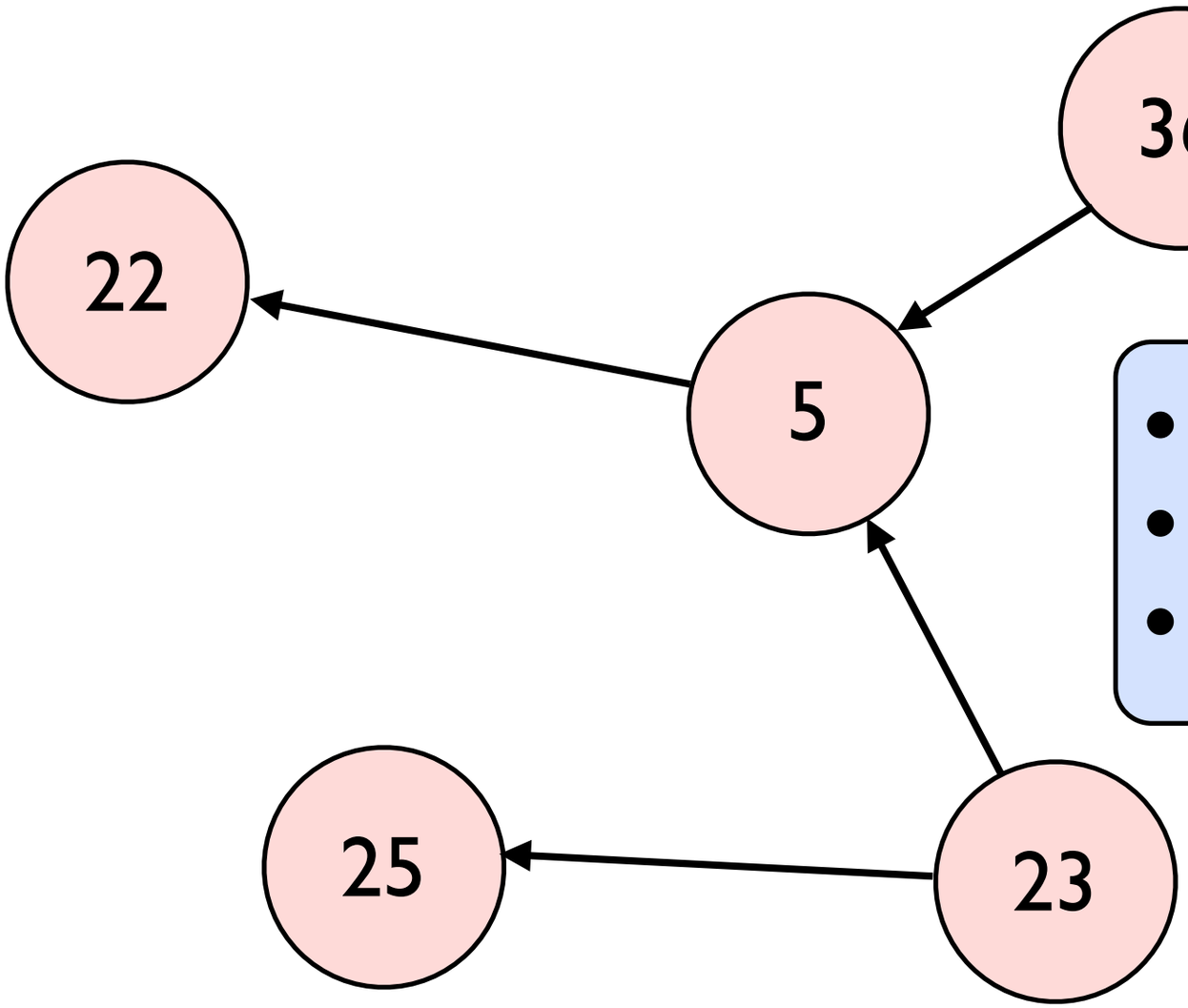
```
x := cons(3,3);
y := cons(4,4);
[x+1] := y;
[y+1] := x;
```



Programs we'll reason about today: graph-manipulating programs (!)



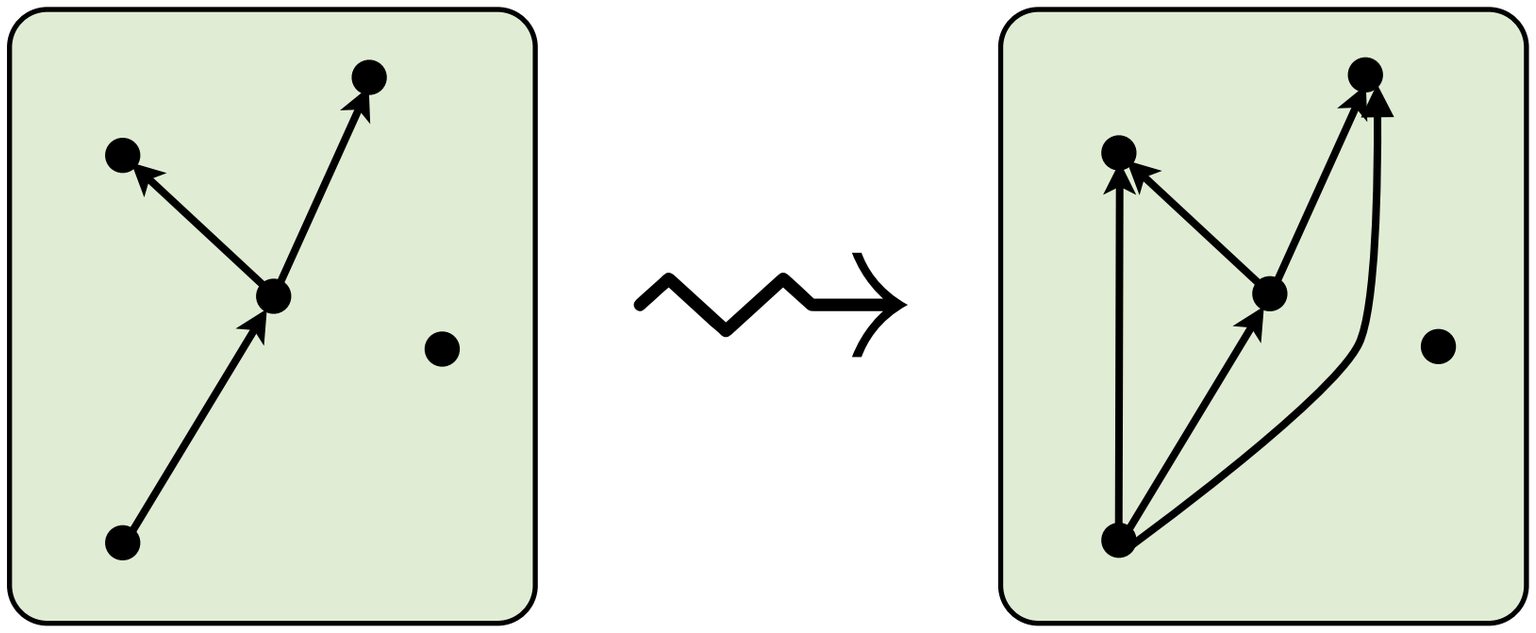
Programs we'll reason about today: graph-manipulating programs (!)



- *What is a "graph manipulation"?*
- *Why reason about graphs?*
- *How do we reason about them?*

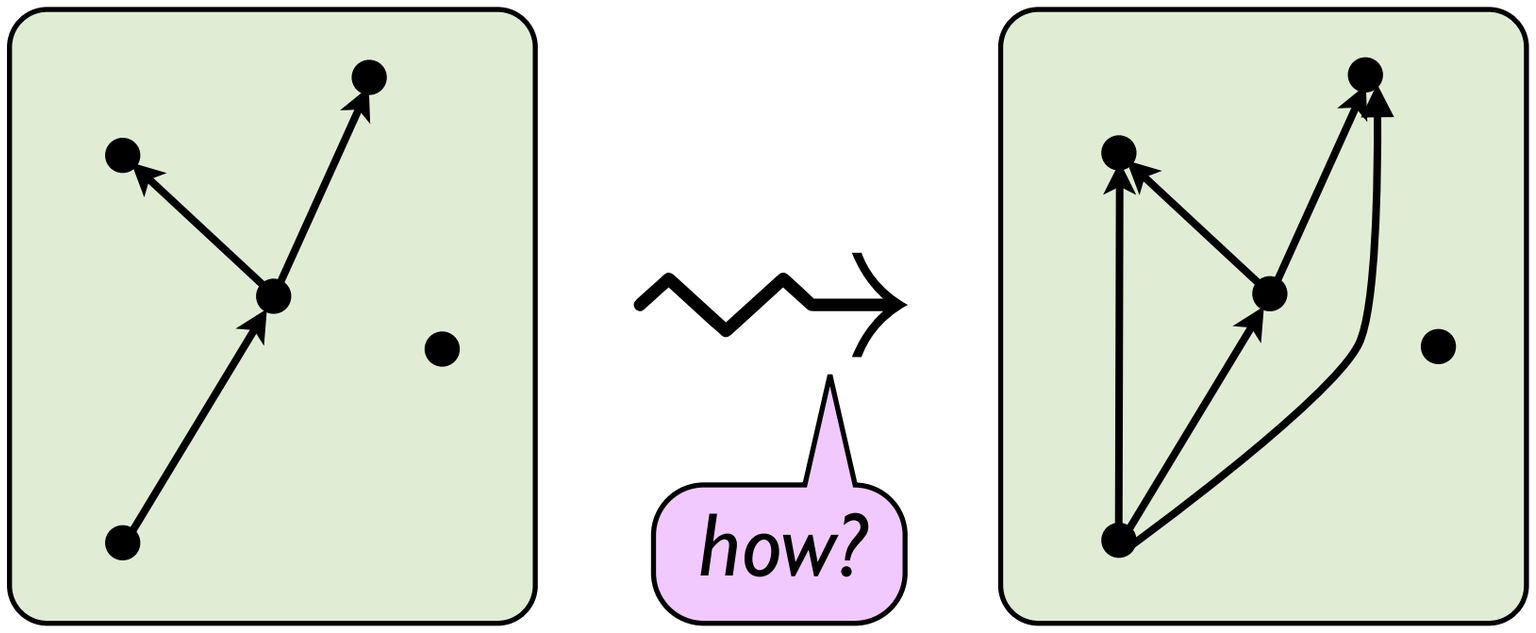
Manipulating graphs?

- creating a new graph out of another algorithmically
  - relabelling (including "marking" nodes/edges)
  - creation/deletion of structure



Manipulating graphs?

- creating a new graph out of another algorithmically
  - relabelling (including "marking" nodes/edges)
  - creation/deletion of structure



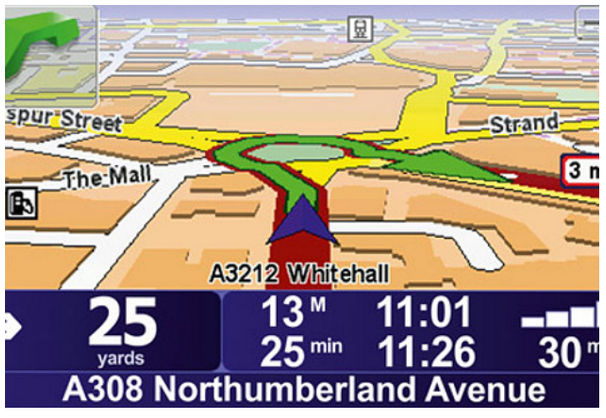
One way to manipulate a graph

- graph as an abstract data type:

  `adjacent(`*G*`,` *v*`,` *w*`)`
  `addEdge(`*G*`,` *v*`,` *w*`)`
  `deleteEdge(`*G*`,` *v*`,` *w*`)`
  etc.

  with the graph data structure represented as
  e.g. an adjacency matrix or adjacency list

- implement graph algorithms
  e.g. Dijkstra's shortest path



- reason about and verify them using separation logic

So, that's all ... ?
(not entirely!)

- can use separation logic in reasoning, but:
  - significant sharing possible in graphs
    => proofs can become complicated

- the beauty of graphs is in their simplicity
  - lose some of this when worrying about representation

- efficiently implementing graph algorithms is not our only aim
  - abstraction facilitates high-level reasoning about conceptually difficult problems

Raise the abstraction!

- we will use graph transformation as a computational abstraction

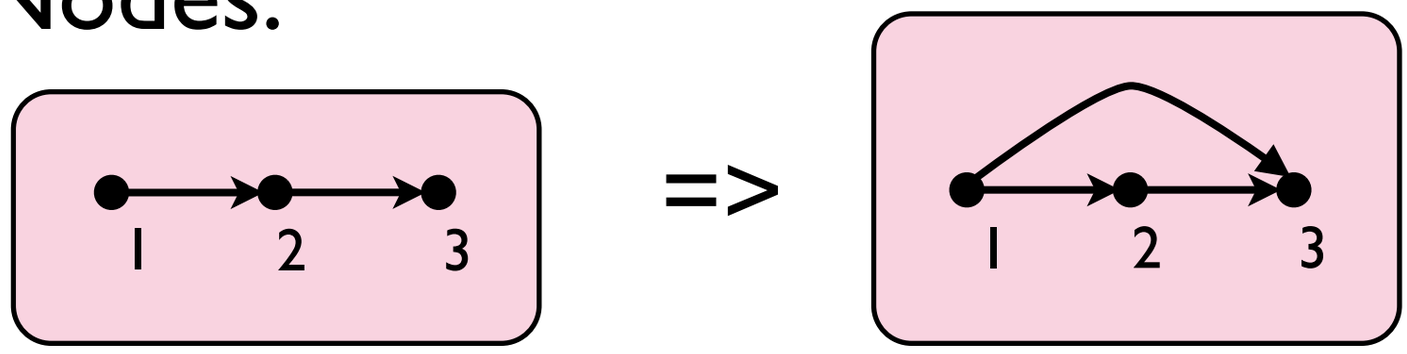
- program states are graphs (in the mathematical sense)

- computational steps are applications of rules

  - akin to Chomsky string-rewriting rules, but for graphs

Example

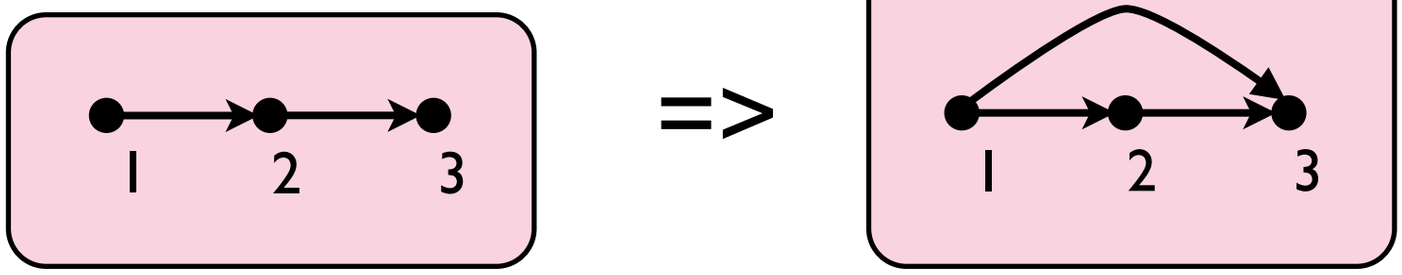
linkNodes:



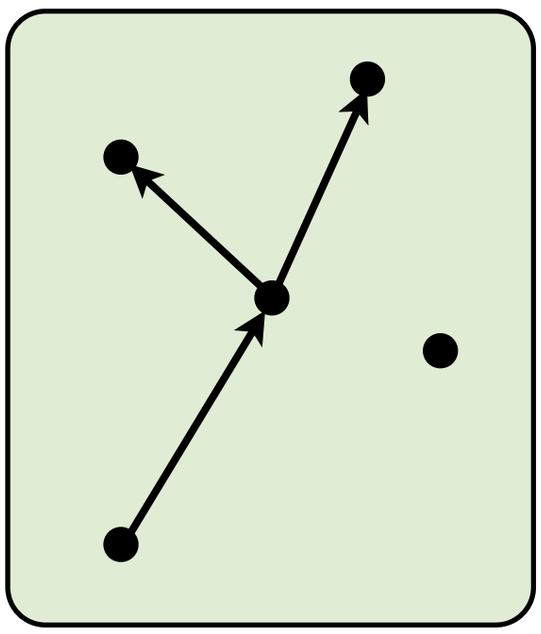
where not edge(1,3)

Example

linkNodes:

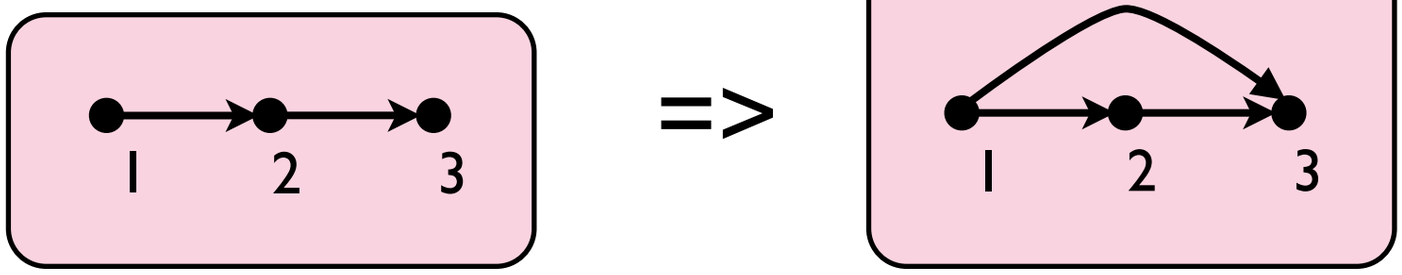


where not edge(1,3)

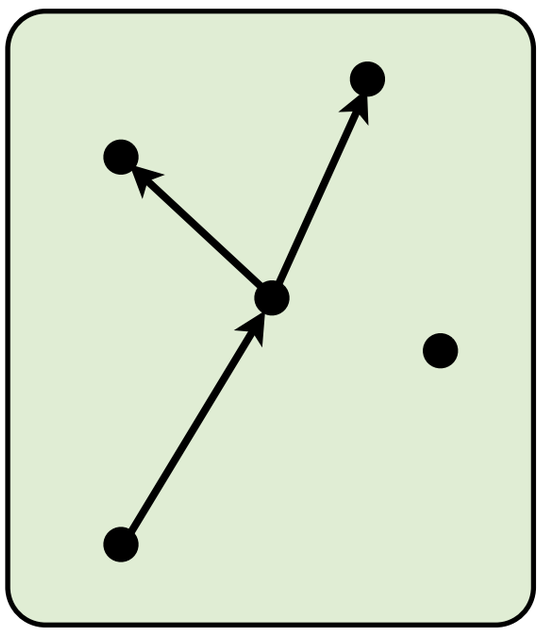


Example

linkNodes:



where not edge(1,3)

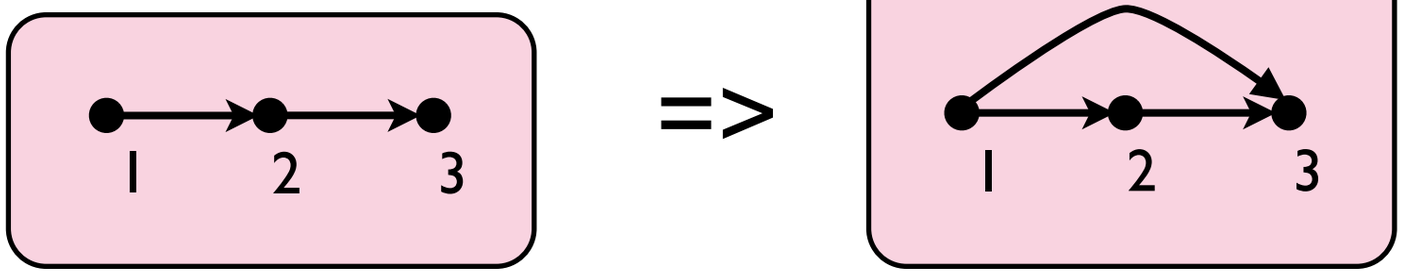


=>

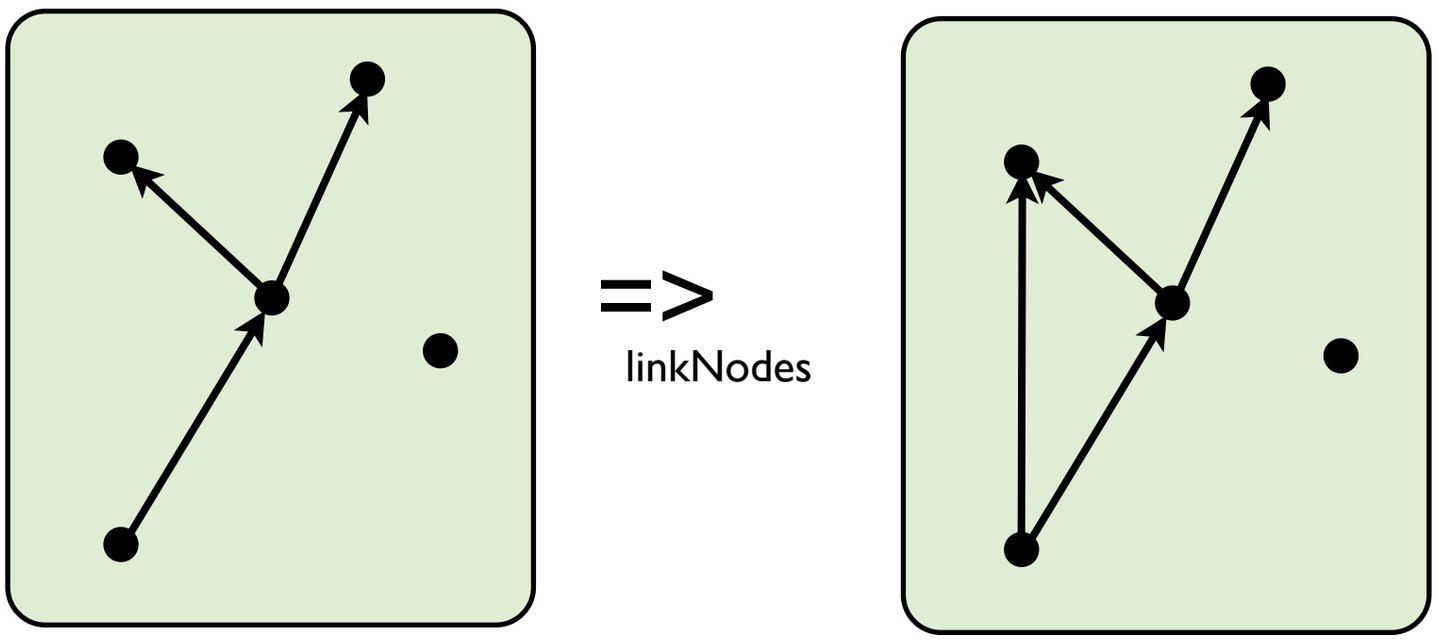
linkNodes

Example

linkNodes:



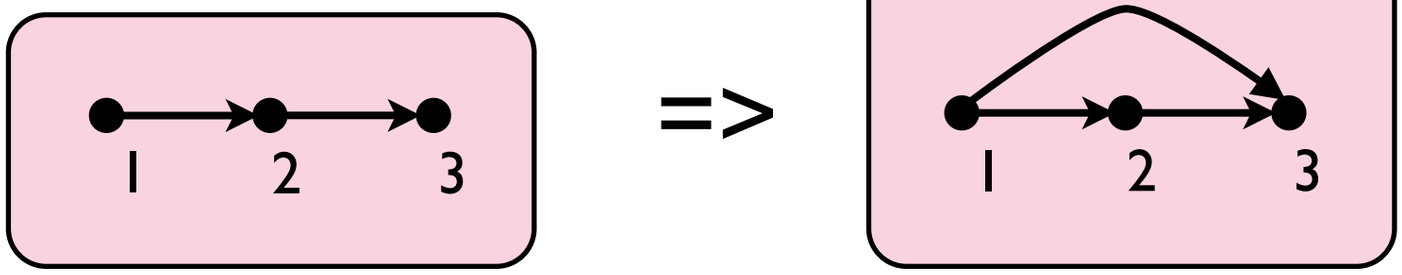
where not edge(1,3)



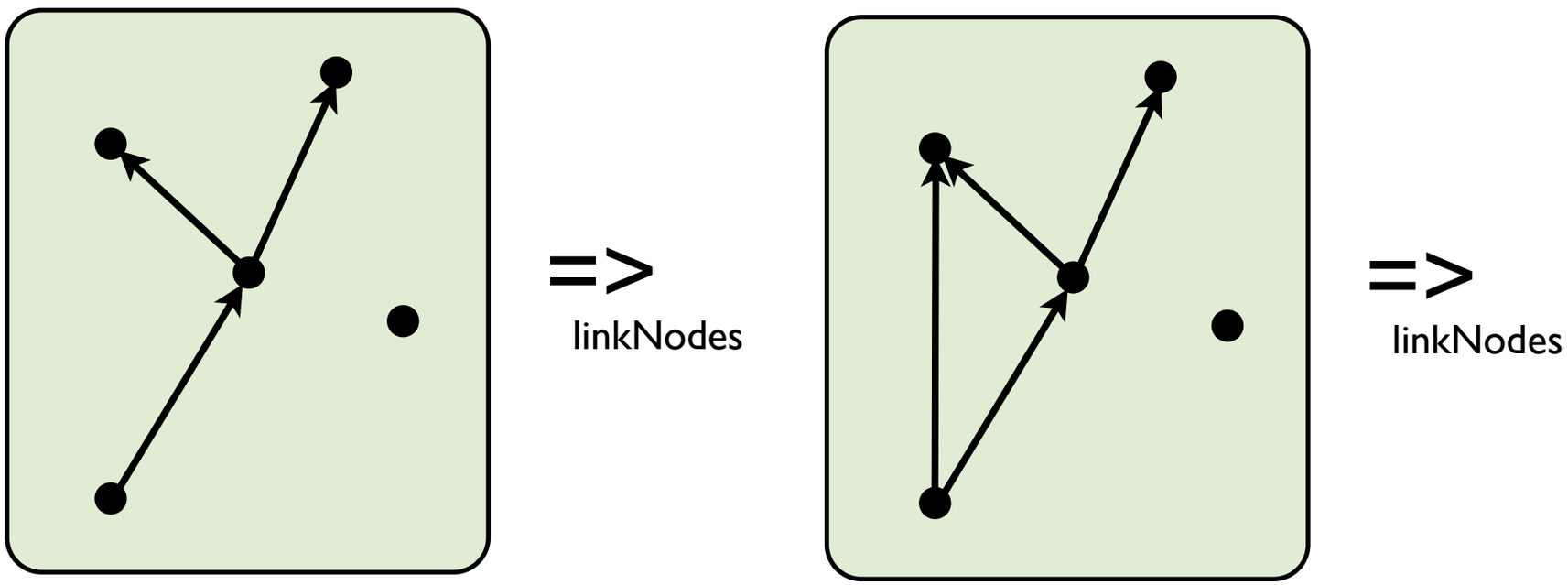
nondeterministic!

Example

linkNodes:



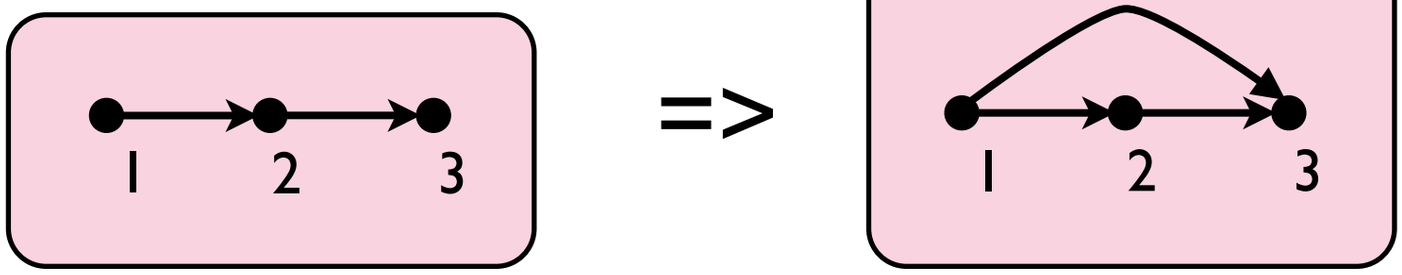
where not edge(1,3)



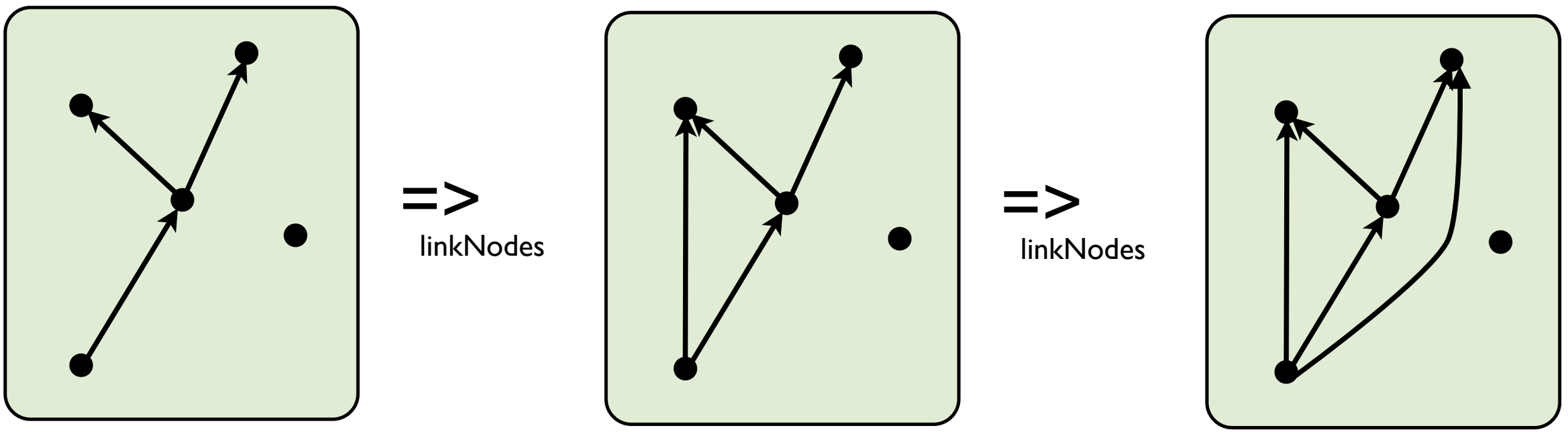
nondeterministic!

Example

linkNodes:



where not edge(1,3)



nondeterministic!

A few of the application areas of graph transformation in CS:

- graph reduction in functional programming languages

- model-driven software development; semantics of UML

- checking shape safety of pointer manipulations

- visual modelling of structure/attribute-changing systems
  - e.g. a rule in a "mobile" system (Pennemann 09)

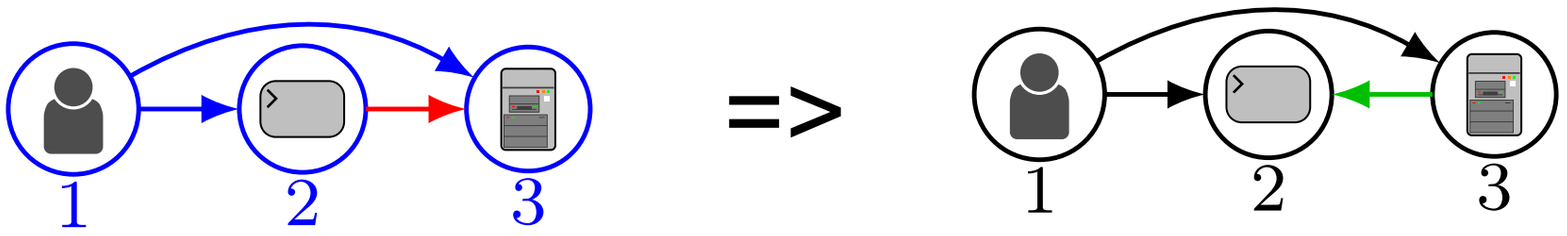
A few of the application areas of graph transformation in CS:

- graph reduction in functional programming languages

- model-driven software development; semantics of UML

- checking shape safety of pointer manipulations

- visual modelling of structure/attribute-changing systems
  - e.g. a rule in a "mobile" system (Pennemann 09)



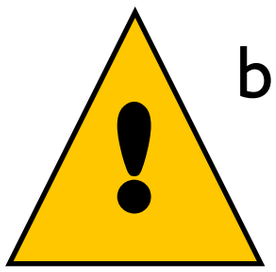
Modelling is only half the story

- modelling problems as graphs and graph transformation rules is only "half the story"
- such visualisations aid in our understanding
  - intuitively express the relations between entities

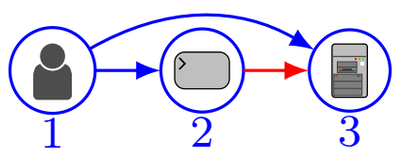
but the use of such techniques alone does not guarantee correctness

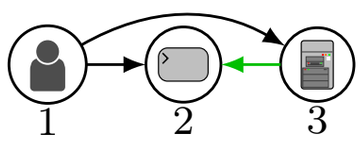
Modelling is only half the story

- modelling problems as graphs and graph transformation rules is only "half the story"
- such visualisations aid in our understanding
  - intuitively express the relations between entities



but the use of such techniques alone does not guarantee correctness

{ *pre ?* } 

=> 

{ *post ?* }

Verifying graph transformations

Ehrig et al.

- a comprehensive theory for graph transformation has been developed since the 1970s
  - based on notions from category theory
  - (only an informal presentation today)

- a basis for sound formal reasoning and verification

- but verification research in the community only gained momentum in the last decade

  - model checking approaches (Rensink, Varró, König, ...)
  - weakest preconditions (Habel & Pennemann)
  - Hoare logic and attributes (Poskitt & Plump)

Verifying graph transformations

- a comprehensive theory for graph transformation has been developed since the 1970s
  - based on notions from category theory
  - (only an informal presentation today)

Ehrig et al.

- a basis for sound formal reasoning and verification

- but verification research in the community only gained momentum in the last decade

  - model checking approaches (Rensink, Varró, König, ...)
  - weakest preconditions (Habel & Pennemann)
  - **Hoare logic and attributes** (Poskitt & Plump)

Next on the agenda

(1) a programming language for graphs

(2) an assertion language for graphs

(3) Hoare-style reasoning about graph transformation

(4) program proofs

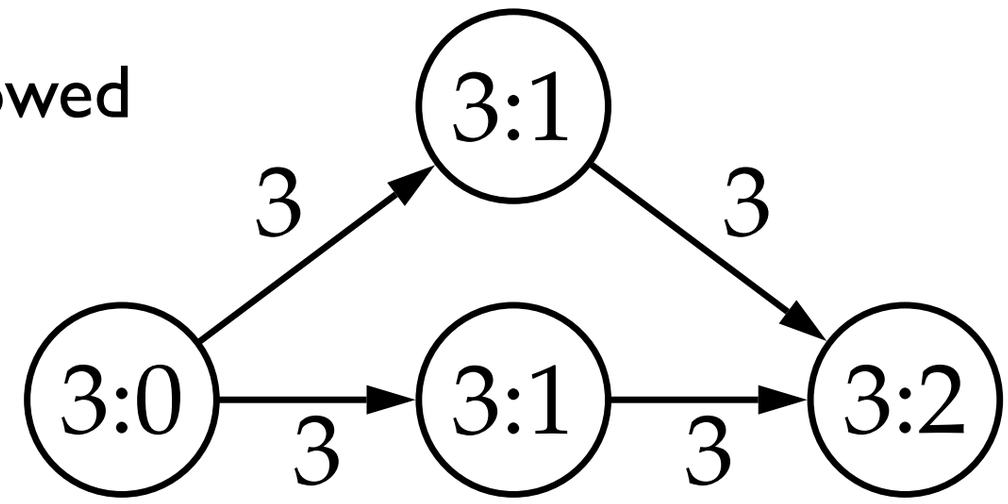
A program state is a graph
(and only a graph)

- not a store, not a heap, not any kind of mapping from variables to data

- just a graph

- label alphabet: (sequences of) integers and strings

- parallel edges and loops allowed



A program state is a graph
(and only a graph)

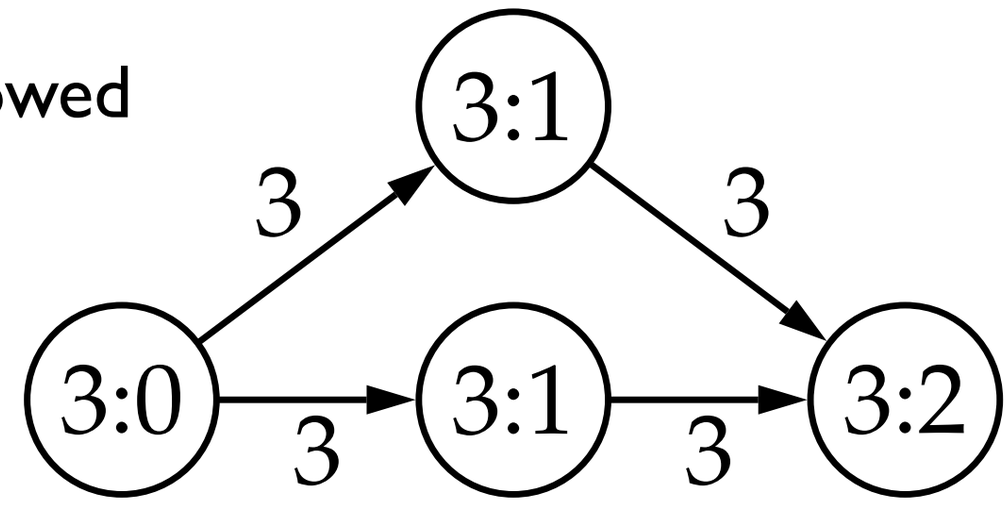
- not a store, not a heap, not any kind of mapping from variables to data

- just a graph

- label alphabet: (sequences of) integers and strings

- parallel edges and loops allowed

what about variables, counters, ... ?



A programming language based on graph transformation

- we will follow the syntax and semantics of GP 2
  - for "Graph Programs"
  - other languages: AGG, Fujaba, and GrGen

Plump

- graph programs comprise two components:

(1) a set of graph transformation rules

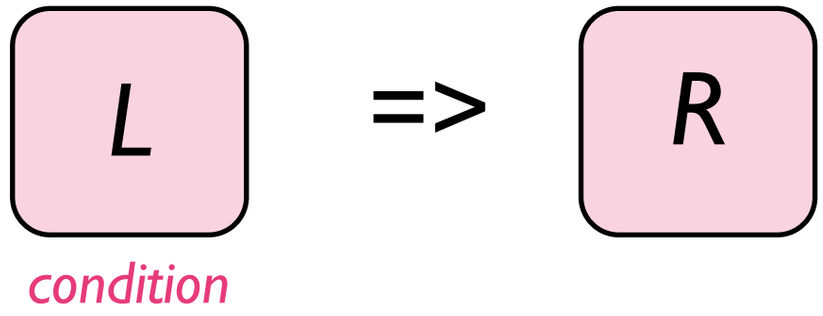
(2) a command sequence informing their application

Graph transformation rules

- rules comprise two graphs:

  (1) a left-hand side *L*, describing what is to be matched

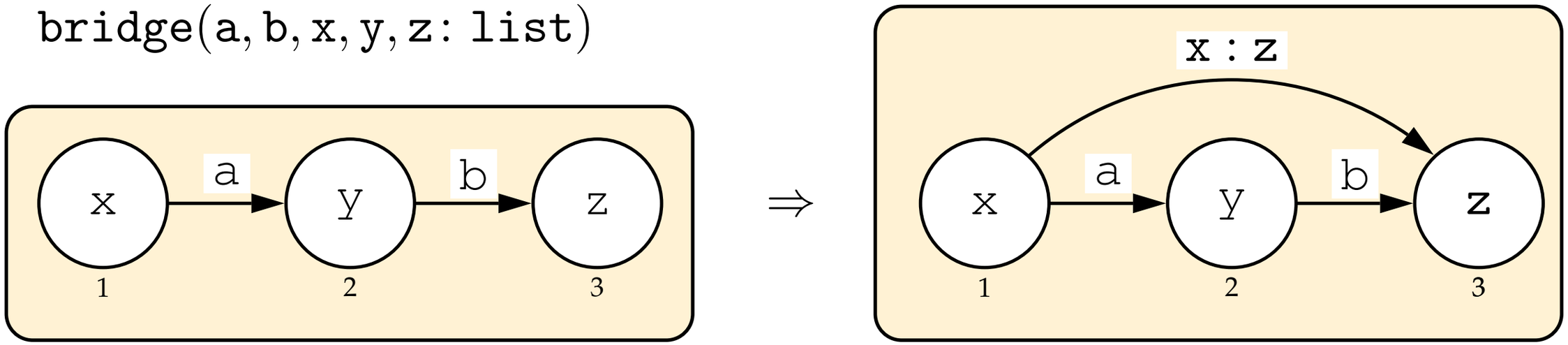
  (2) a right-hand side *R*, describing what to replace the match with



- both *L* and *R* are labelled over expressions
- rule can be equipped with a textual condition
  - expressing relations between labels, nodes

Example

```
bridge(a, b, x, y, z: list)
```

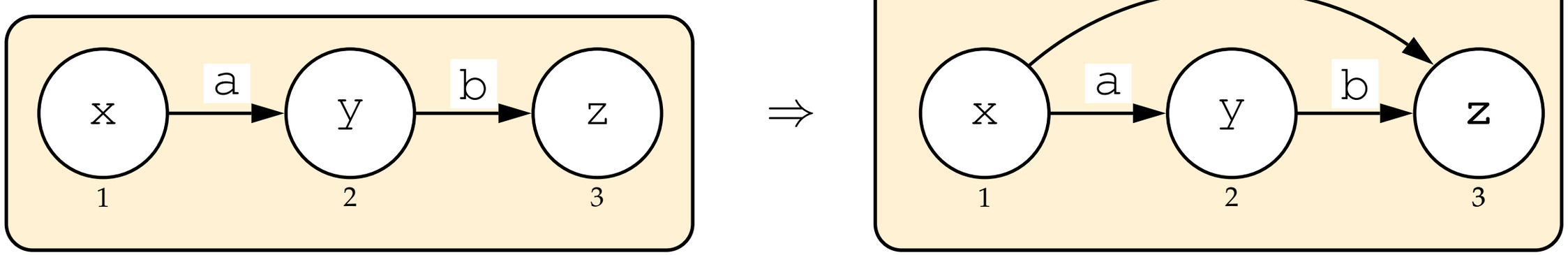



```
where not edge(1, 3)
```

Example

rule name

```
bridge(a, b, x, y, z: list)
```

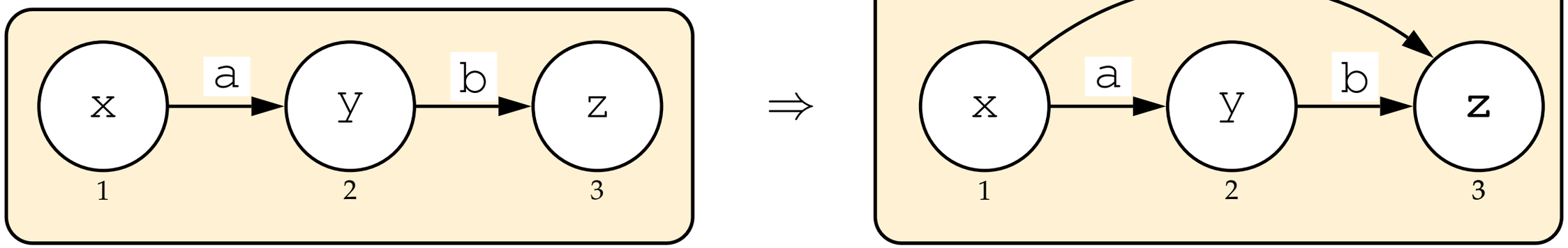



```
where not edge(1, 3)
```

Example

"variables", instantiated during graph matching
(type list = sequences of ints and strings)

```
bridge(a, b, x, y, z: list)
```

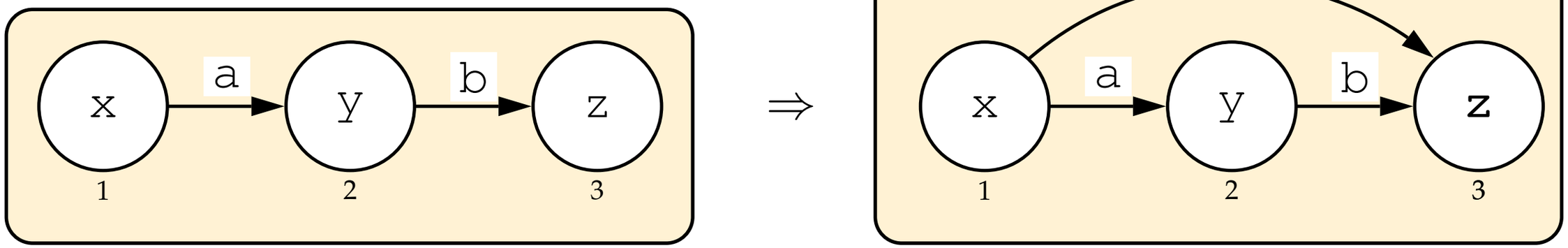



```
where not edge(1, 3)
```

Example

"variables", instantiated during graph matching
(type list = sequences of ints and strings)

```
bridge(a, b, x, y, z: list)
```

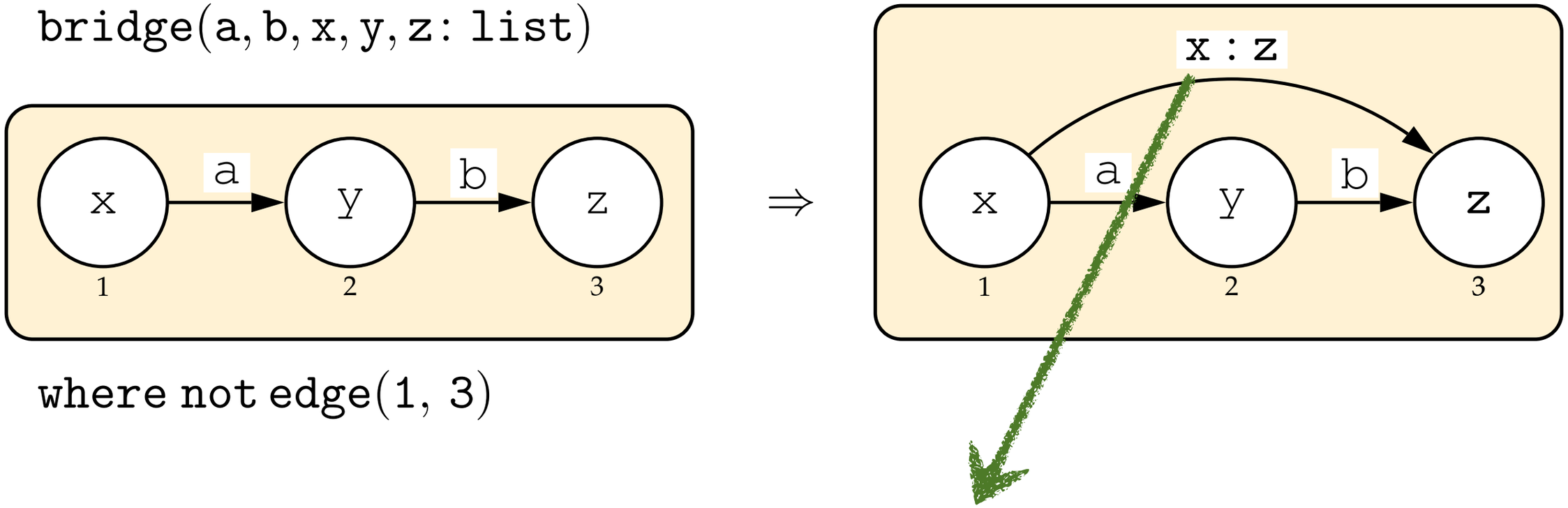



```
where not edge(1, 3)
```

program states (graphs) do not have variables
...but rules do!
Here they are placeholders, not references to a store

Example

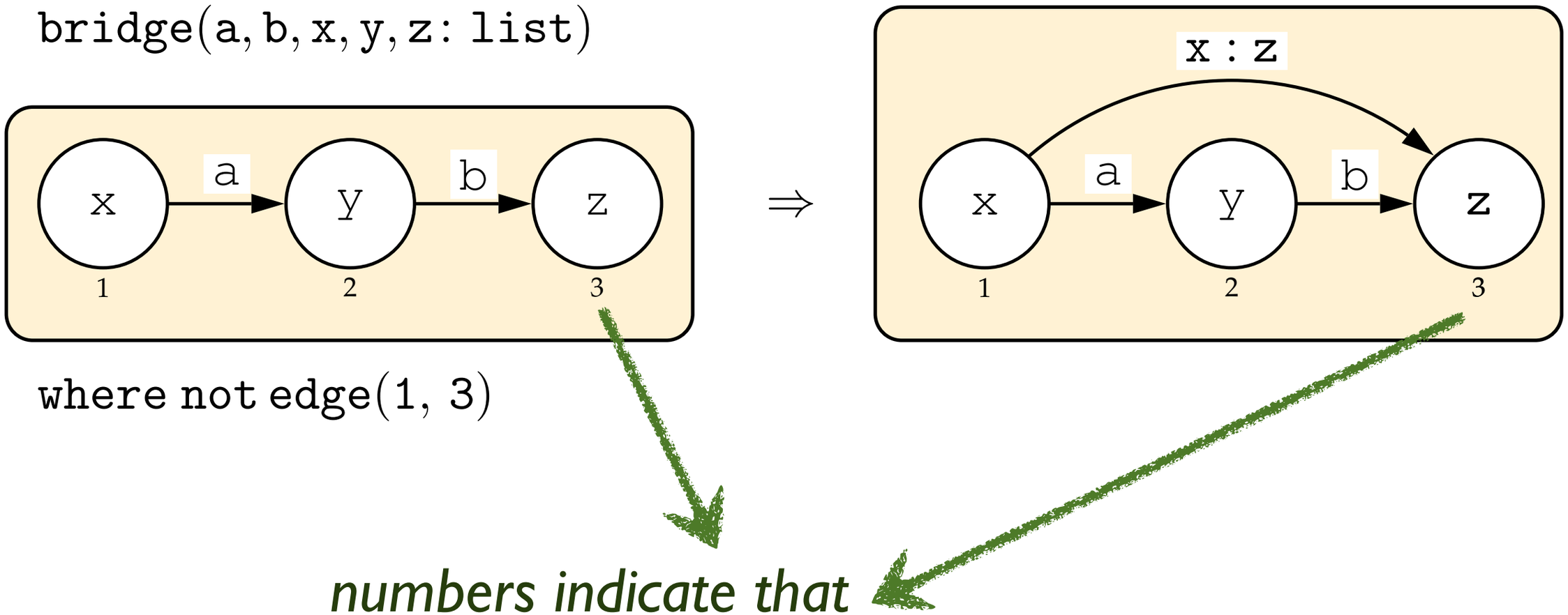
```
bridge(a, b, x, y, z: list)
```




```
where not edge(1, 3)
```

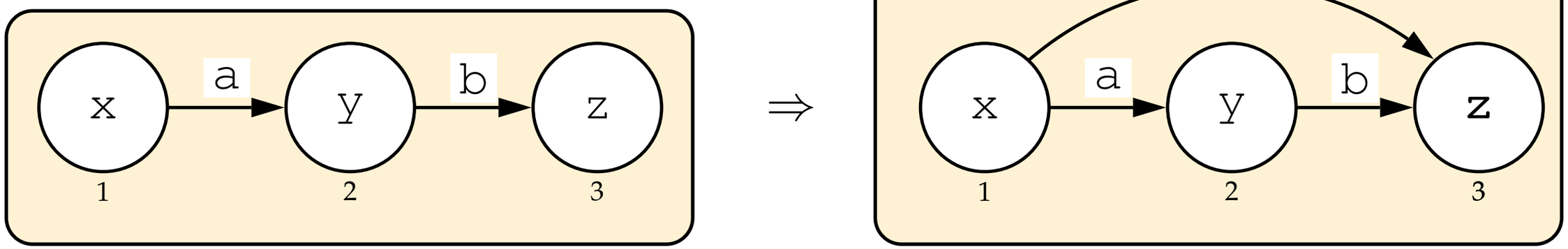
new edge takes the label x' : z'
where x |-> x' and z |-> z'

Example


bridge(a, b, x, y, z: list)
a
b
x
y
z
1
2
3
⇒
x : z
a
b
x
y
z
1
2
3
where not edge(1, 3)
numbers indicate that
the nodes are the same

Example

```
bridge(a, b, x, y, z: list)
```



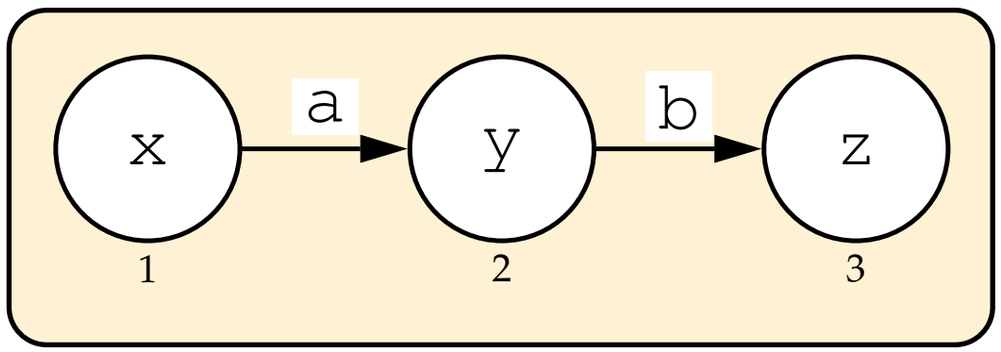

```
where not edge(1, 3)
```

a condition: rule cannot be applied if there is an edge from matched node 1 to matched node 3

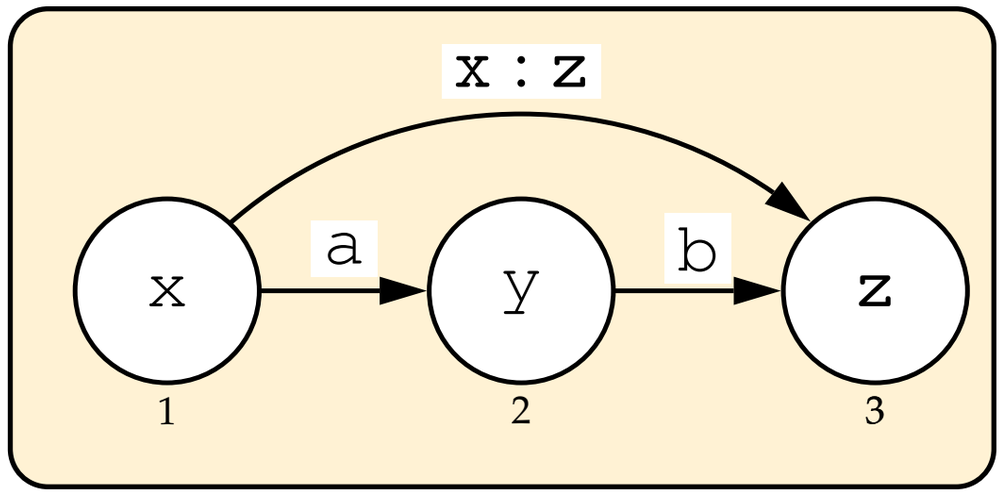
Example rule application

`bridge(a, b, x, y, z: list)`

L

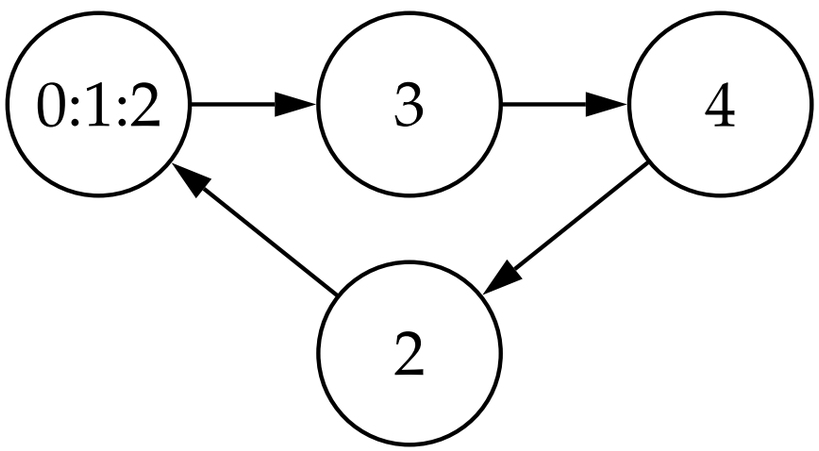


⇒



`where not edge(1, 3)`

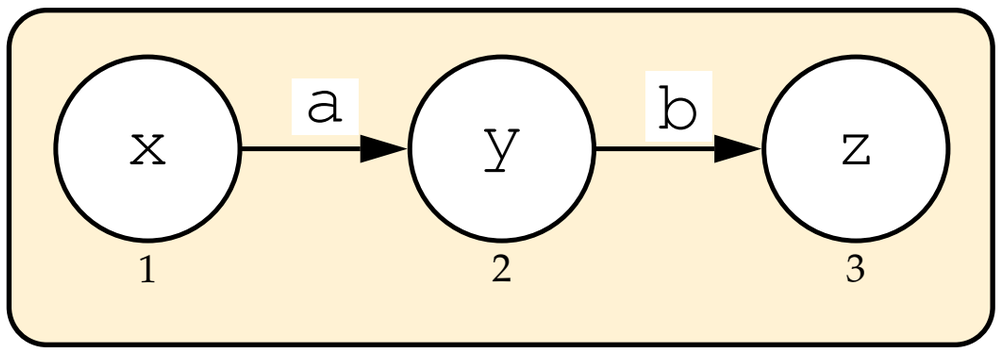
let us apply bridge to this graph



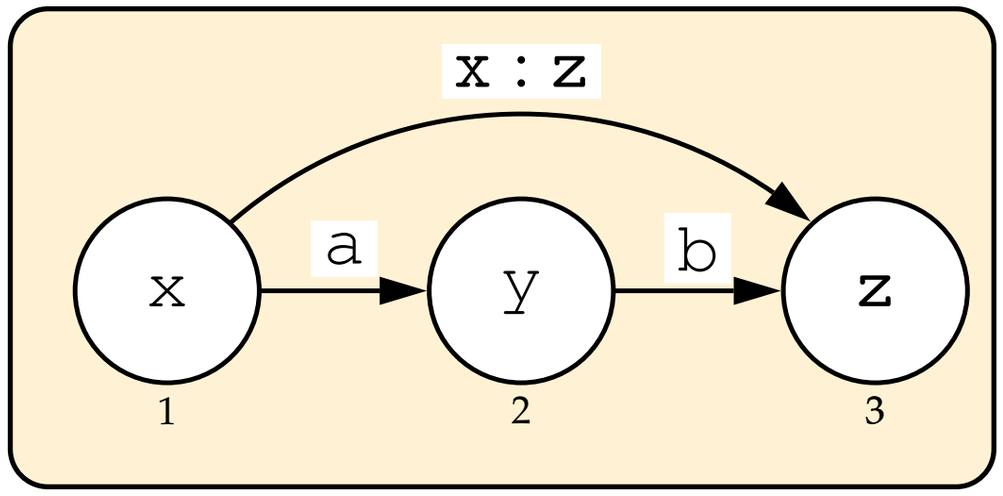
Example rule application

`bridge(a, b, x, y, z: list)`

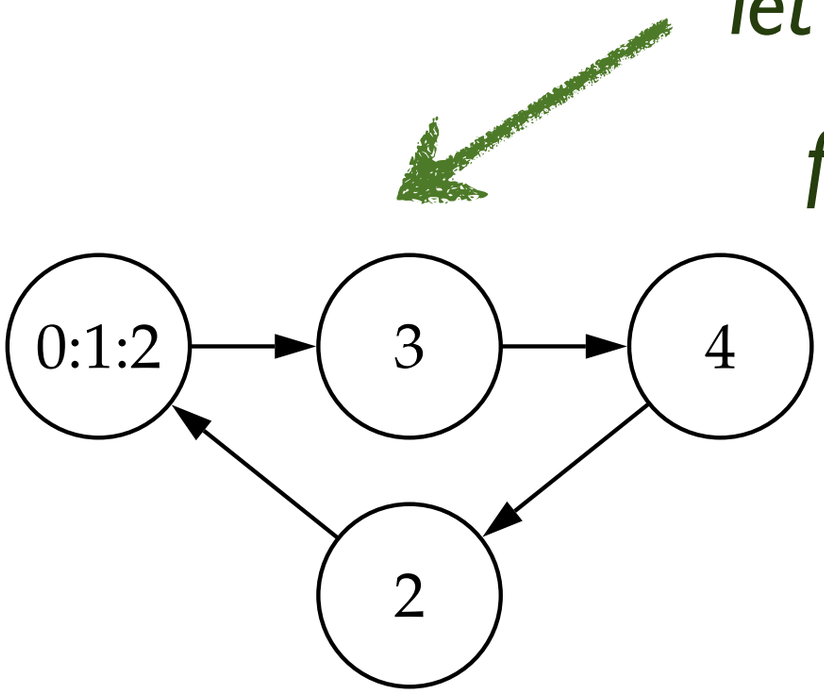
L



⇒



`where not edge(1, 3)`



let us apply bridge to this graph

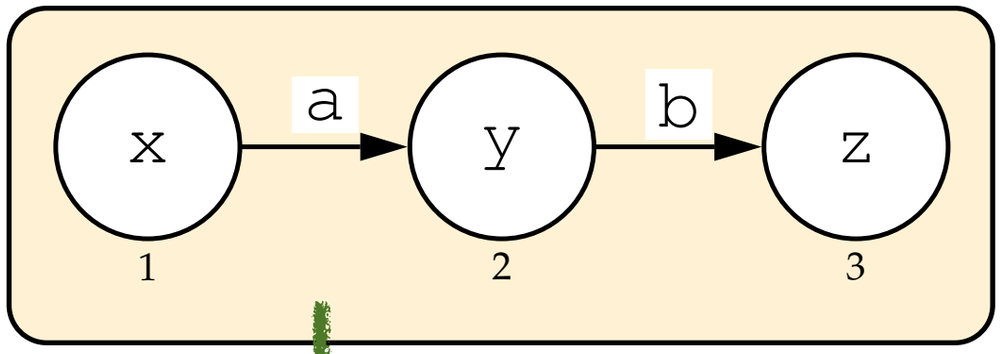
find mapping α: Vars -> Data such that:

(1) there is a "match" for L^{α} in the graph
(2) the condition holds

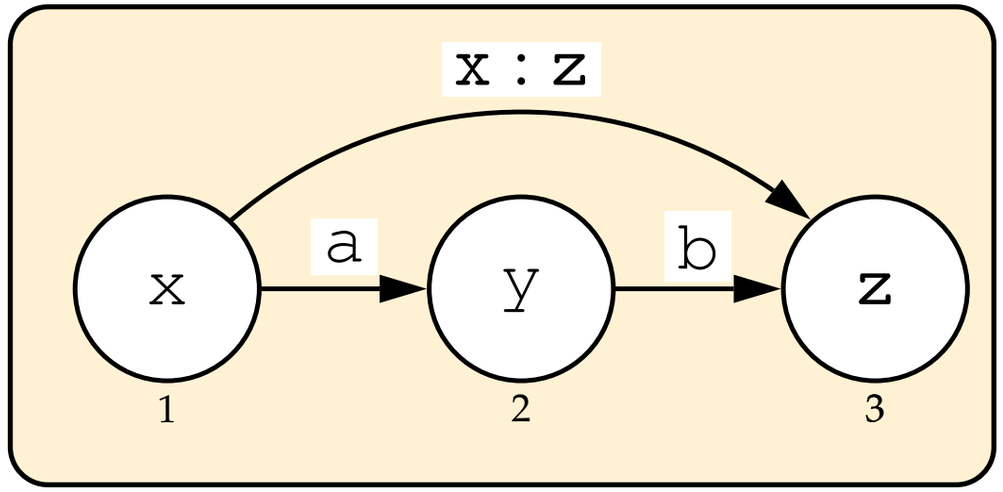
Example rule application

`bridge(a, b, x, y, z: list)`

L



⇒

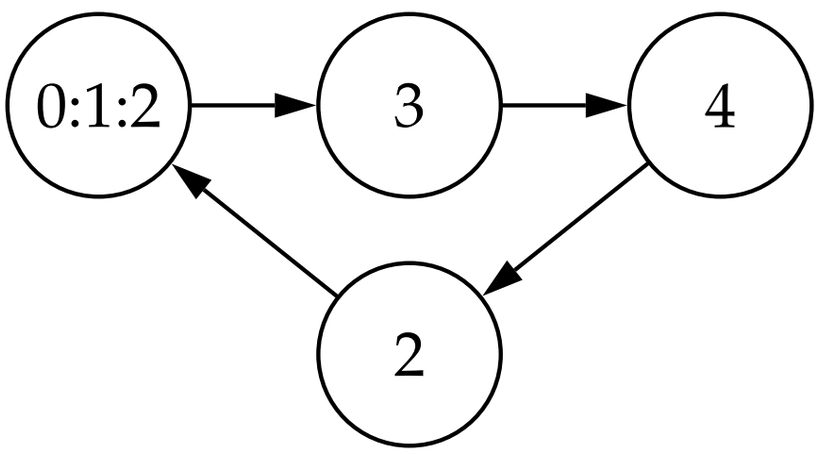


`where not edge(1, 3)`

α

x |-> *0:1:2*
y |-> *3*
z |-> *4*

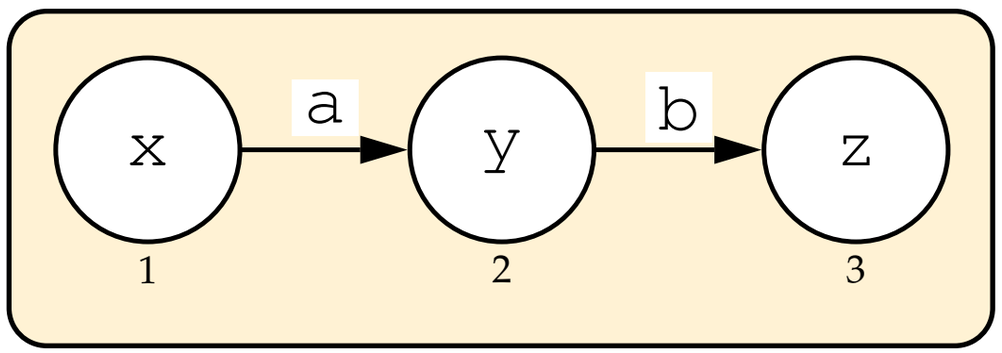
a |-> blank
b |-> blank



Example rule application

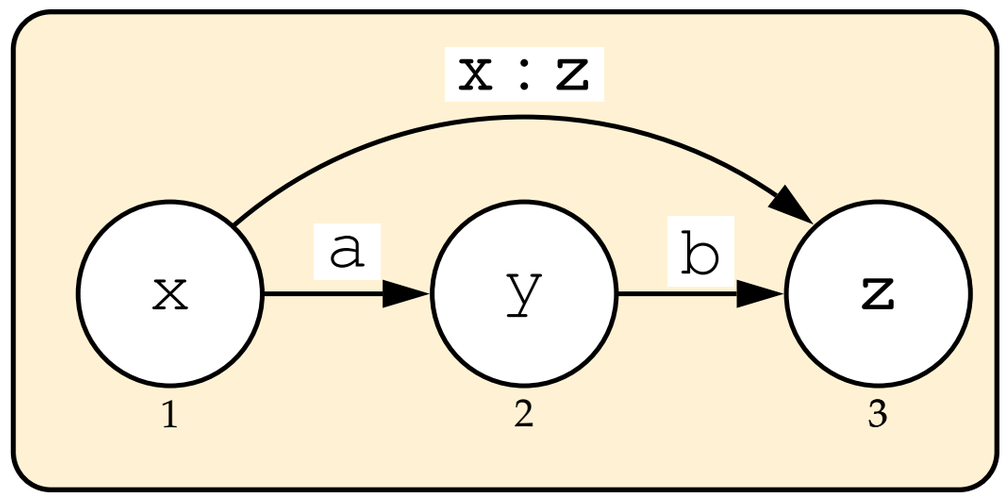
`bridge(a, b, x, y, z: list)`

L

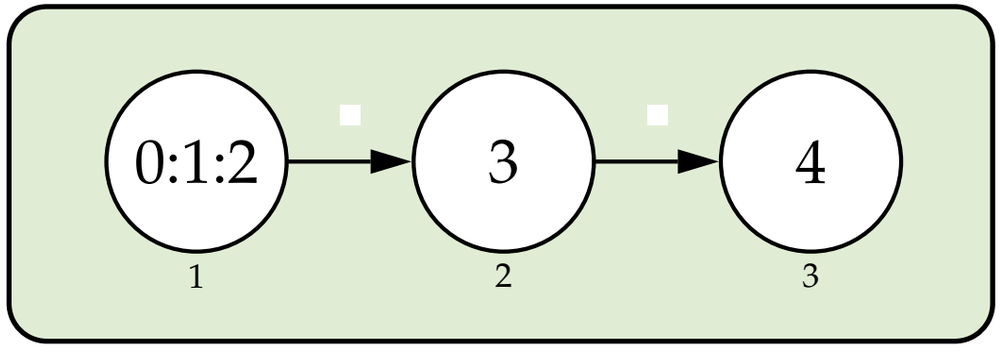


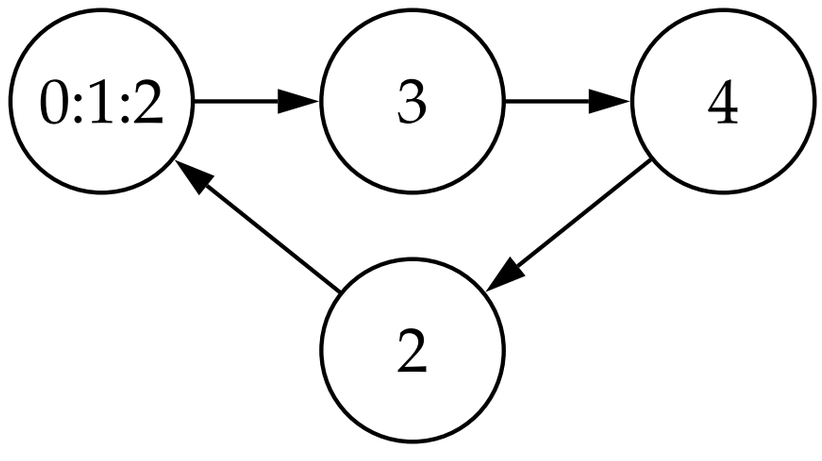
`where not edge(1, 3)`

⇒



L^{α}





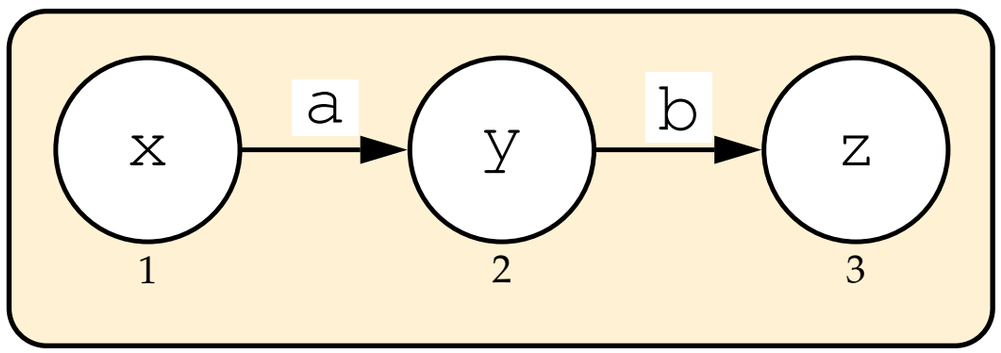
x |-> *0:1:2*
y |-> *3*
z |-> *4*

a |-> blank
b |-> blank

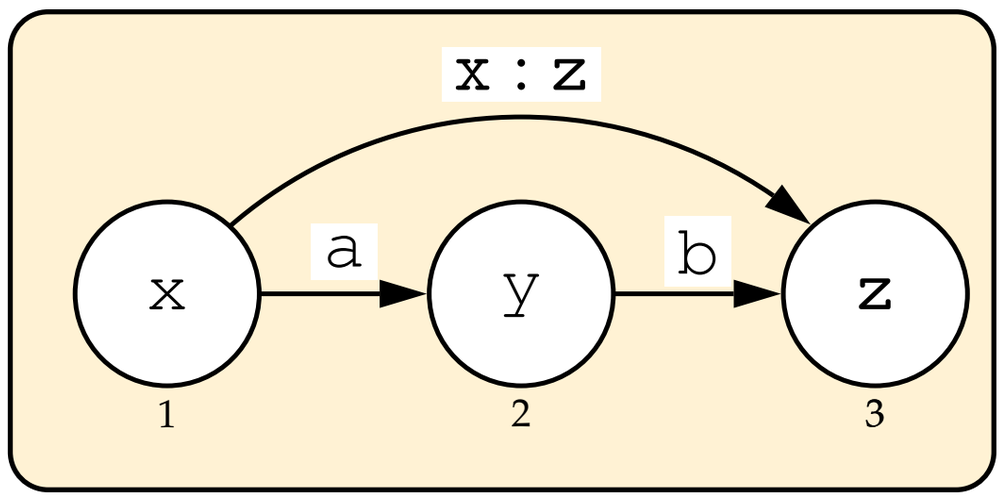
Example rule application

`bridge(a, b, x, y, z: list)`

L

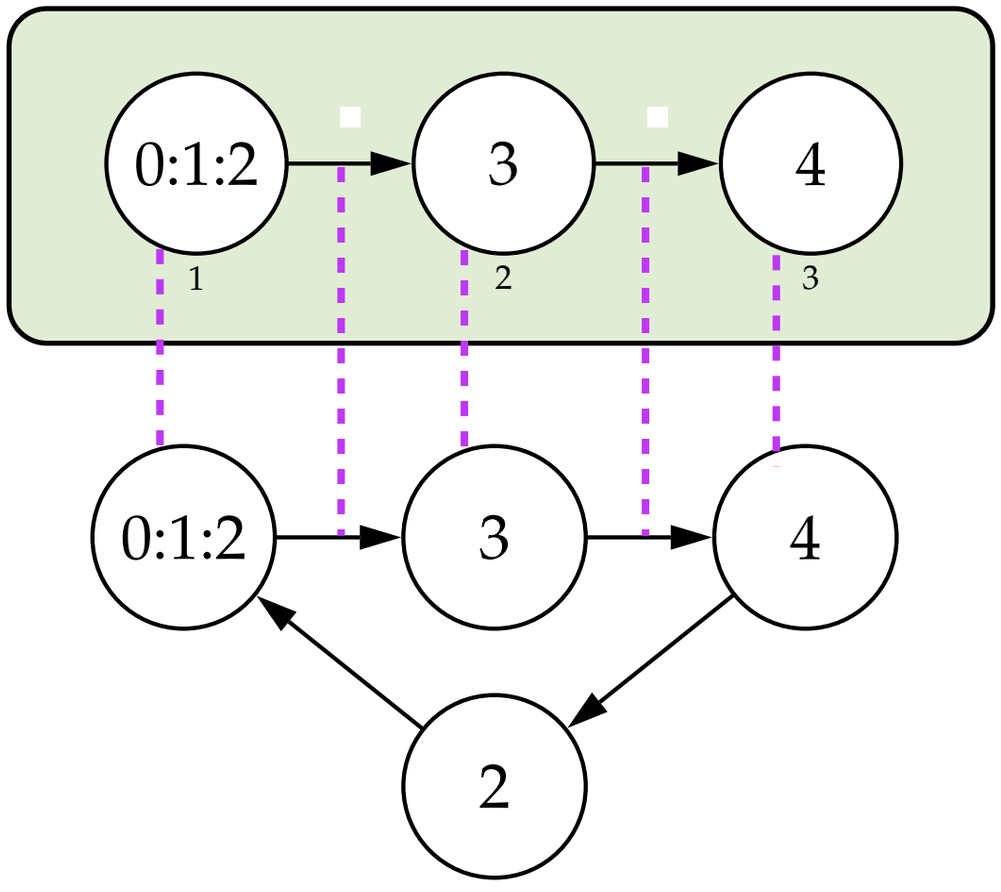


⇒



`where not edge(1, 3)`

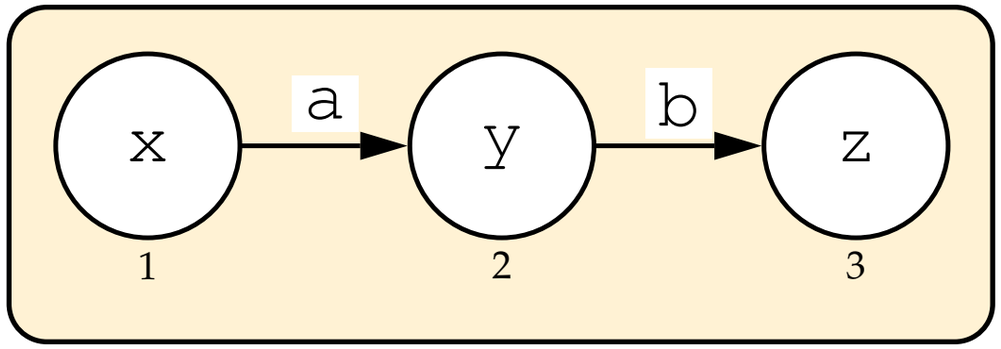
L^{α}



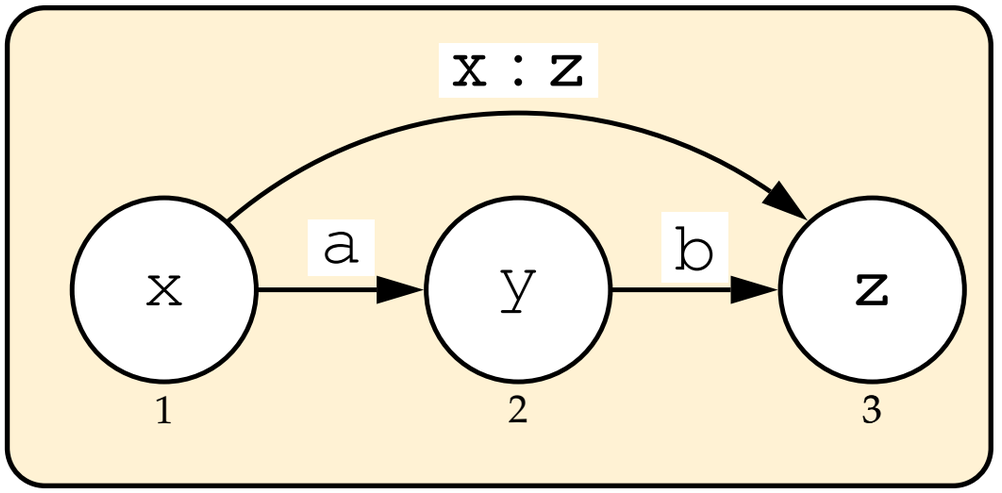
Example rule application

`bridge(a, b, x, y, z: list)`

L

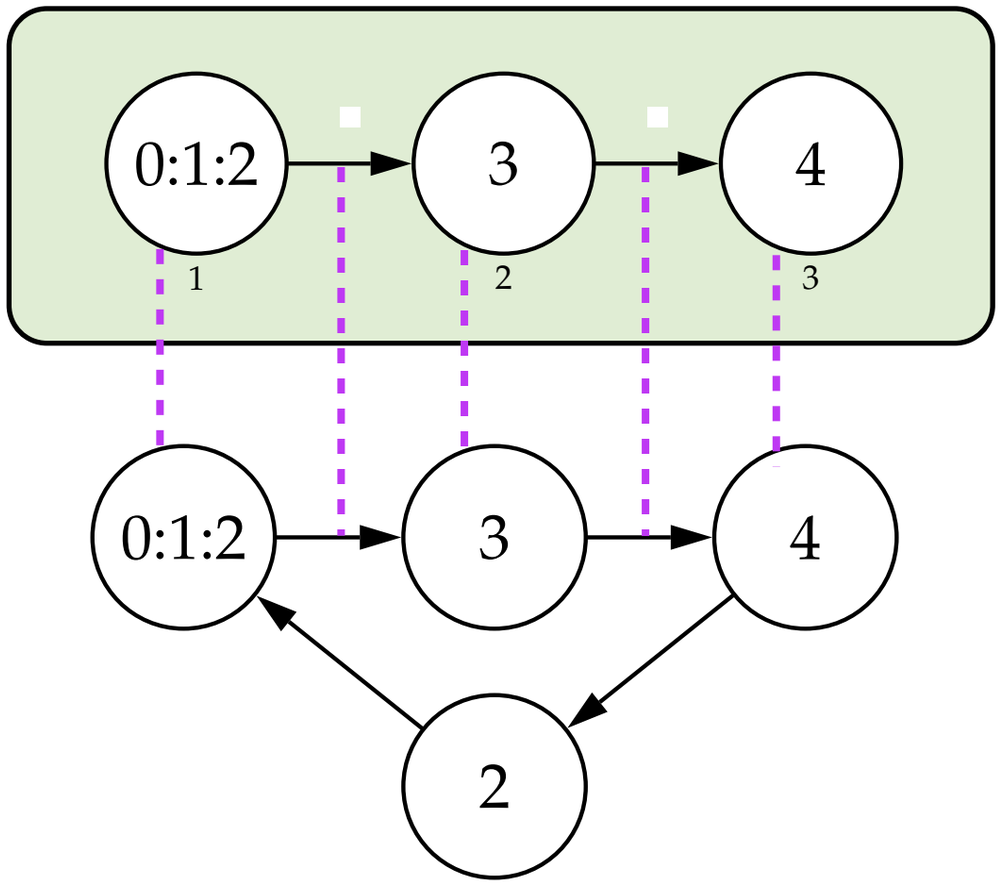


⇒

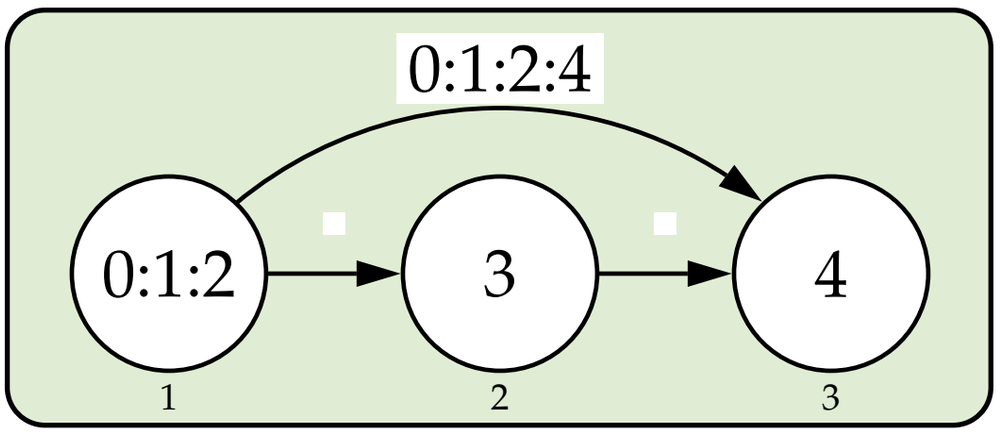


`where not edge(1, 3)`

L^{α}



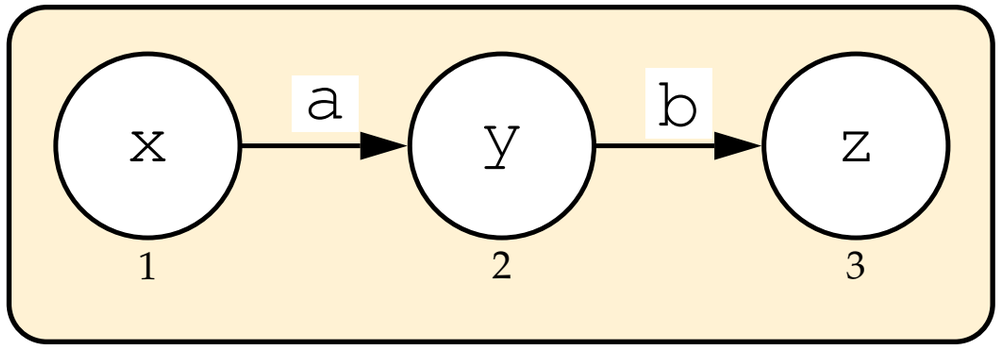
⇒



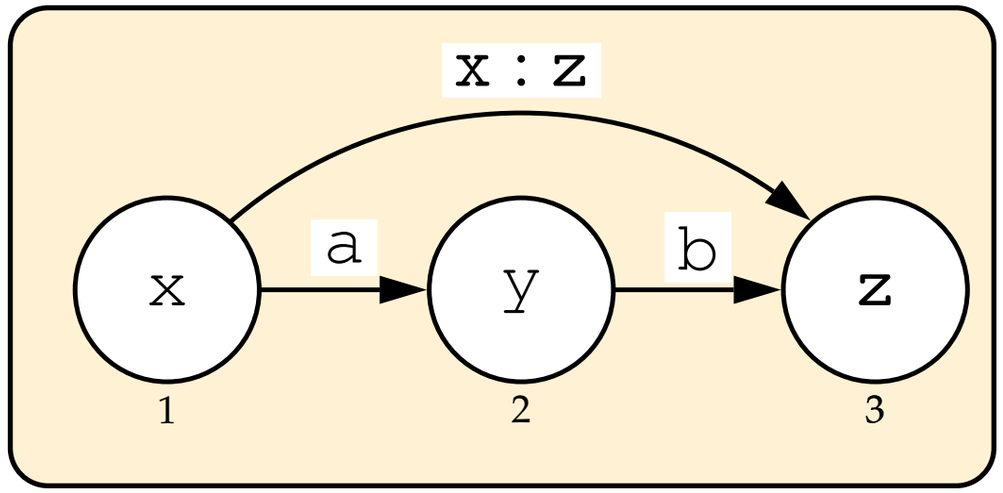
Example rule application

`bridge(a, b, x, y, z: list)`

L

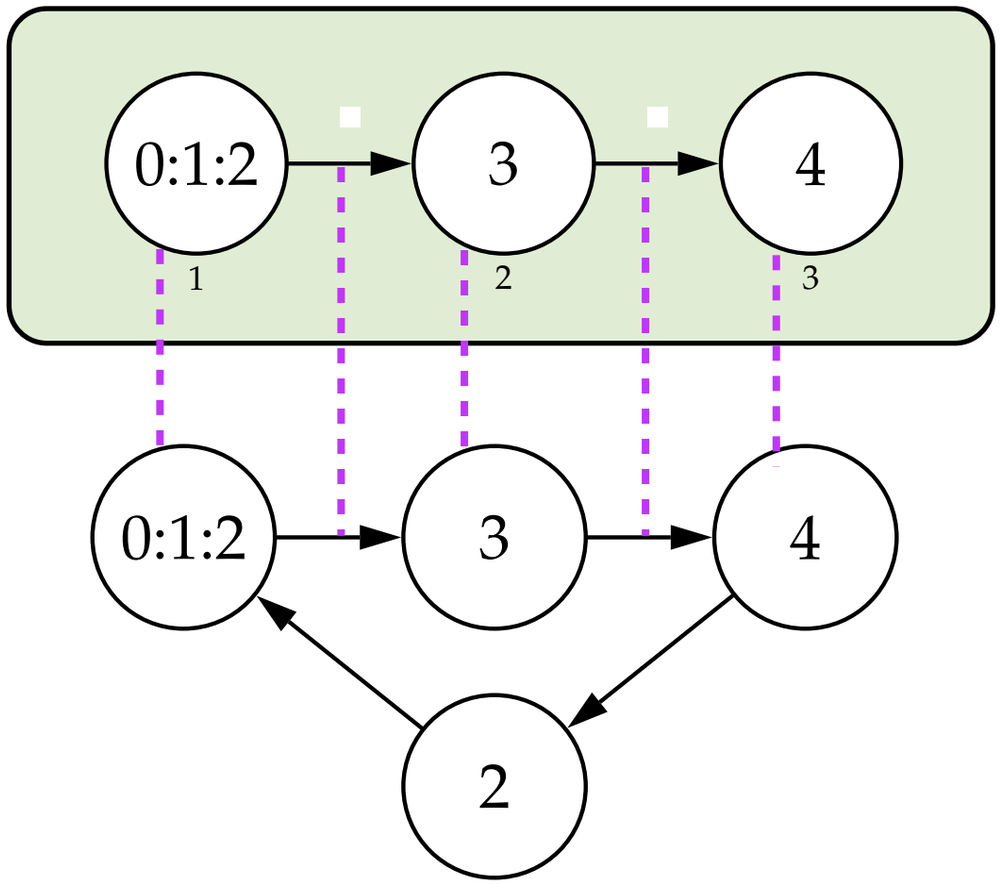


⇒

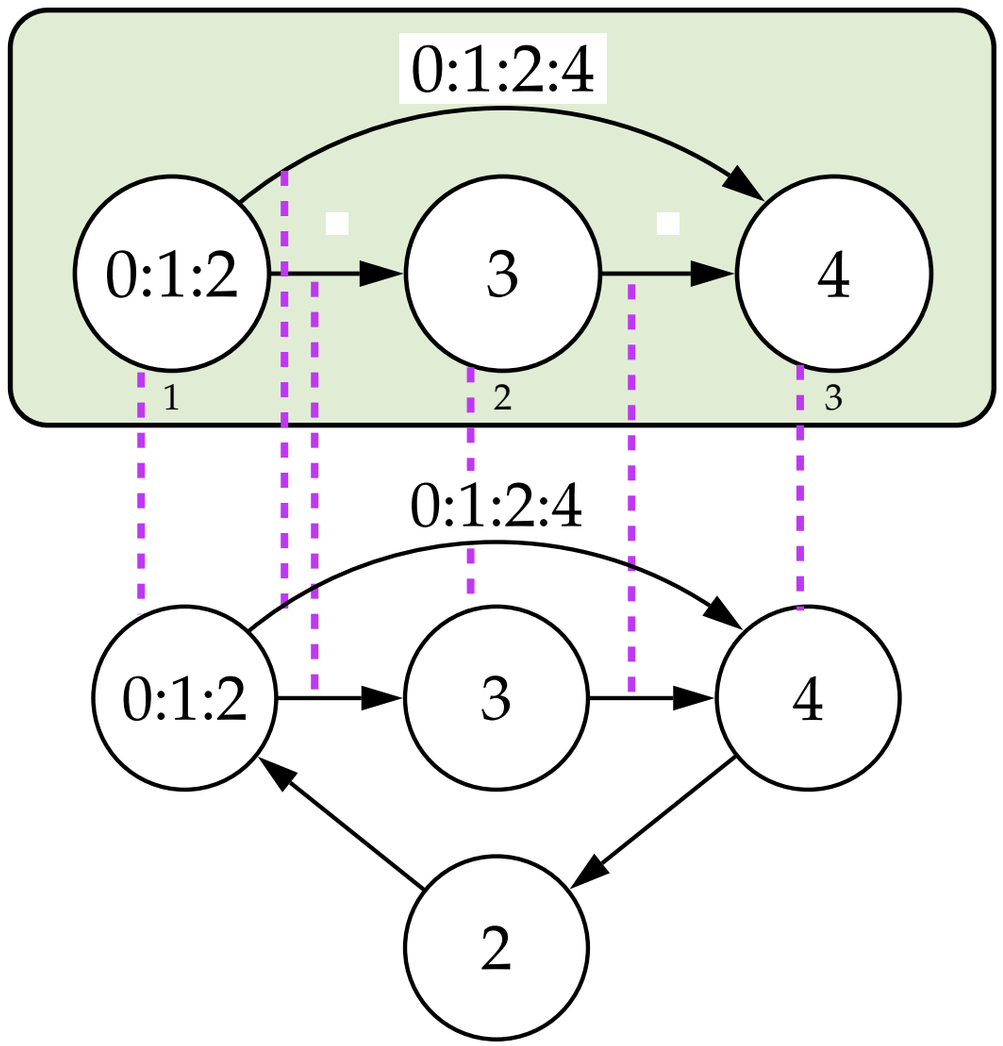


`where not edge(1, 3)`

L^{α}



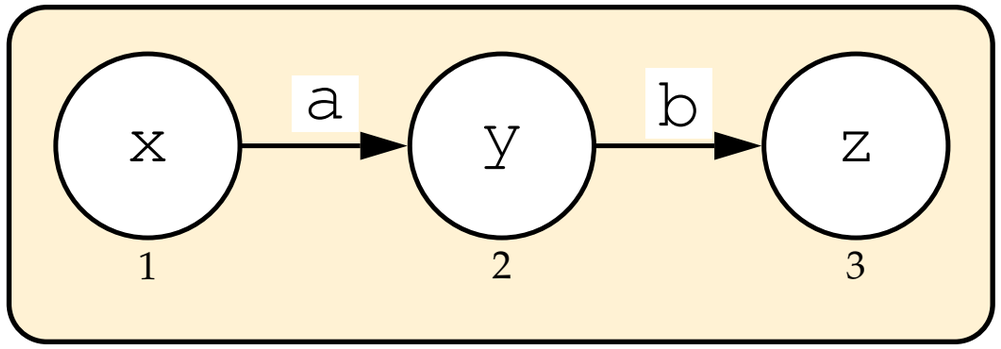
⇒



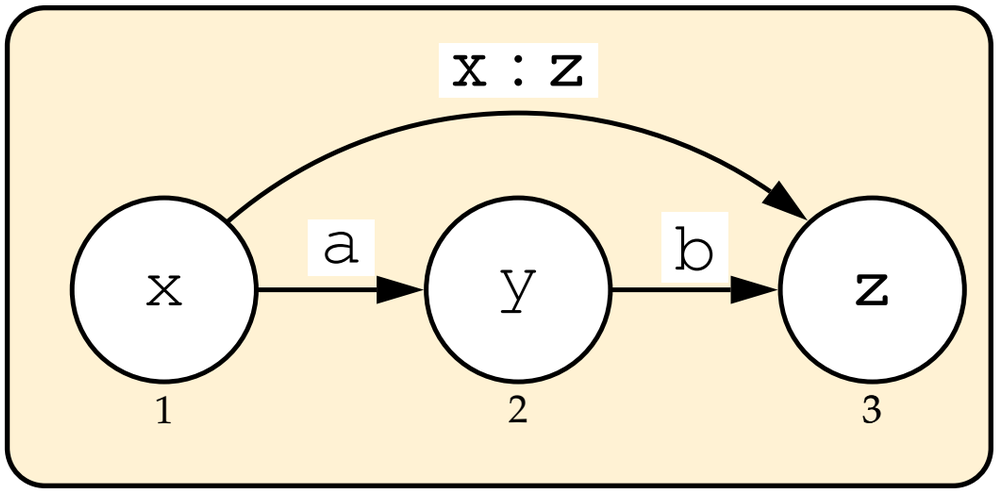
Example rule application

`bridge(a, b, x, y, z: list)`

L

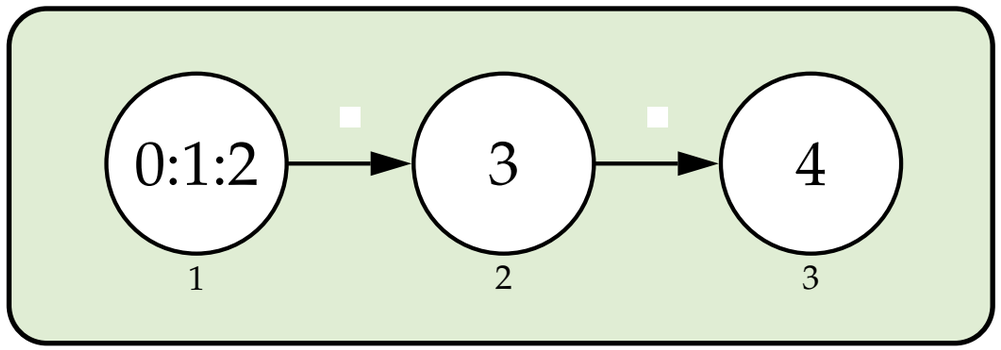


⇒

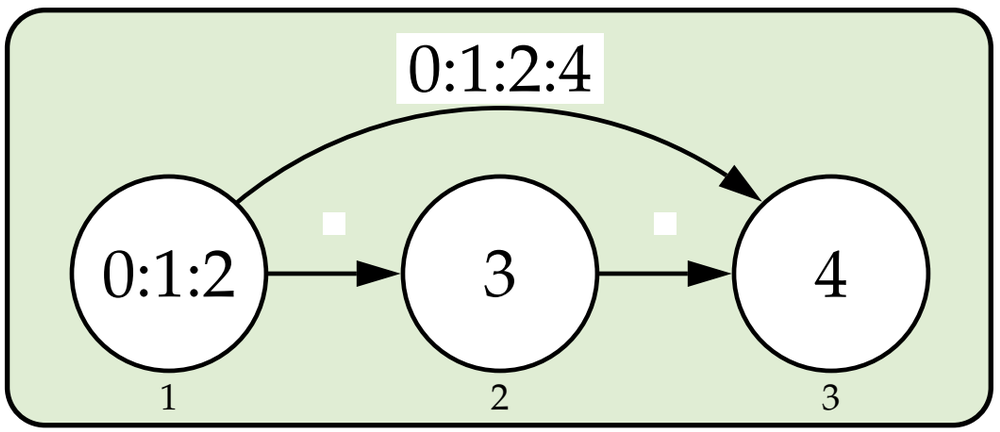


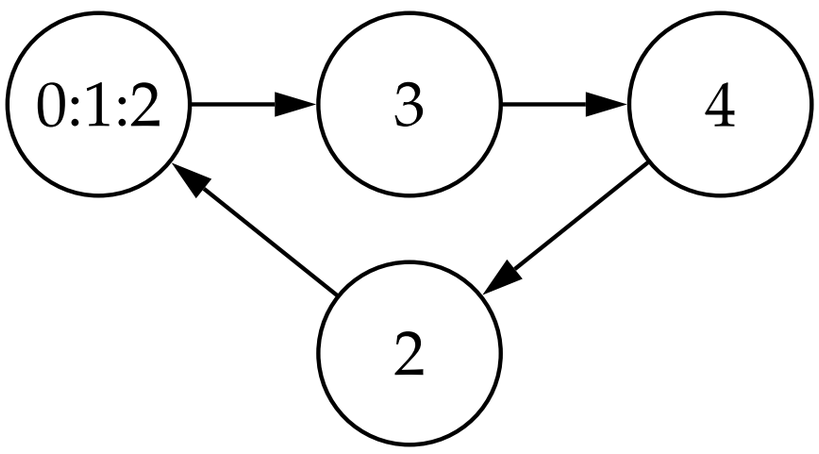
`where not edge(1, 3)`

L^{α}

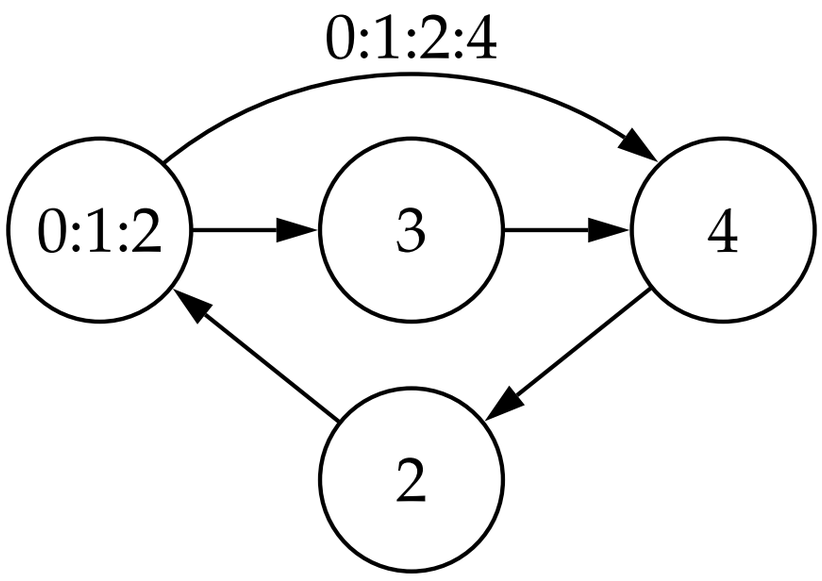


⇒



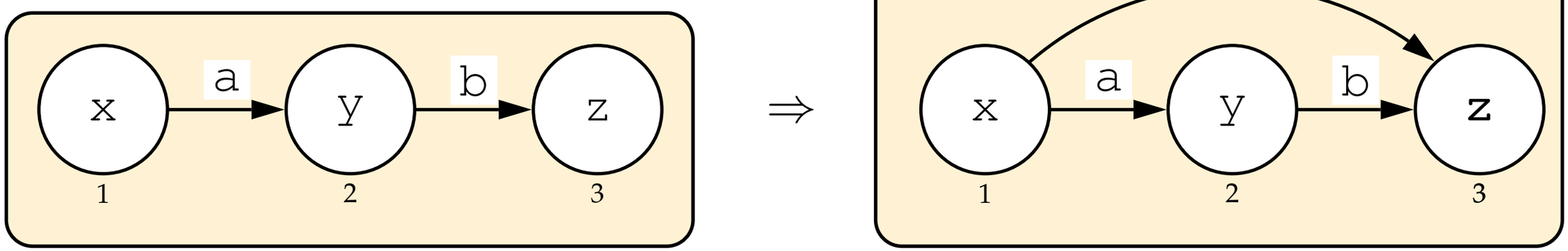


⇒

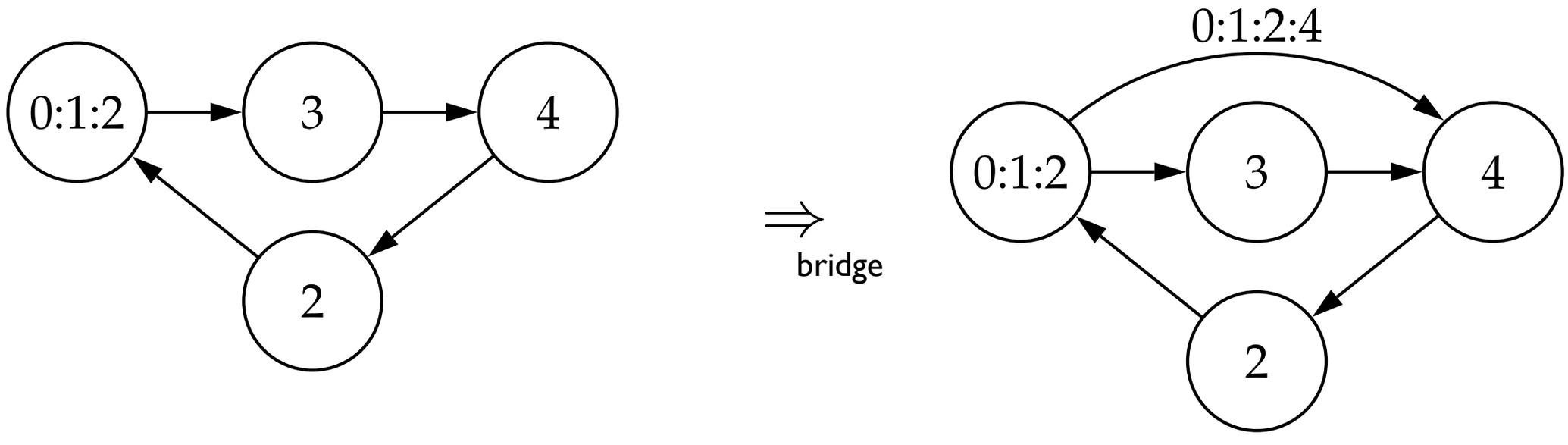


Example rule application

```
bridge(a, b, x, y, z: list)
```

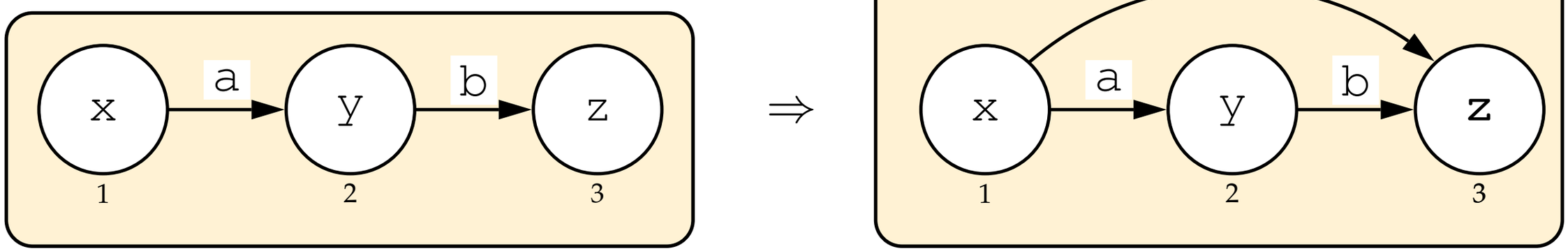



```
where not edge(1, 3)
```

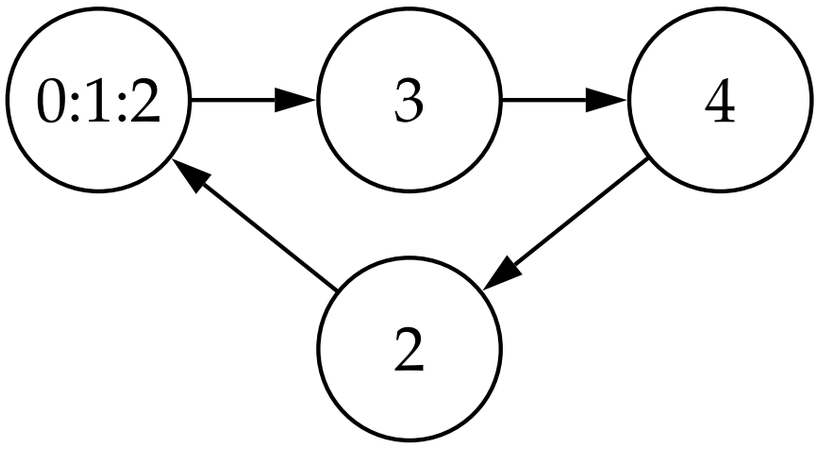


Rule application is nondeterministic

`bridge(a, b, x, y, z: list)`



`where not edge(1, 3)`



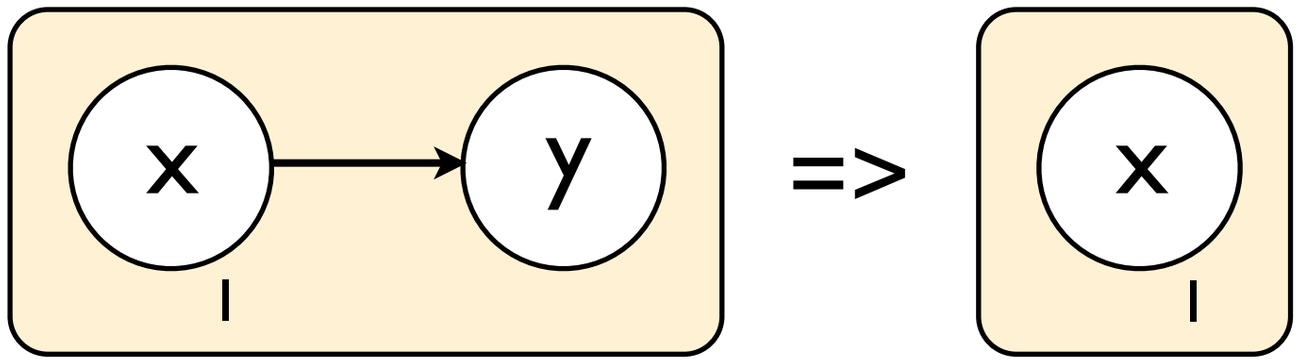
rule application is nondeterministic
- what other graph could result?

Deleting nodes

- we can create nodes/edges and relabel without issue
- we can even delete edges without issue
- can we arbitrarily delete nodes?

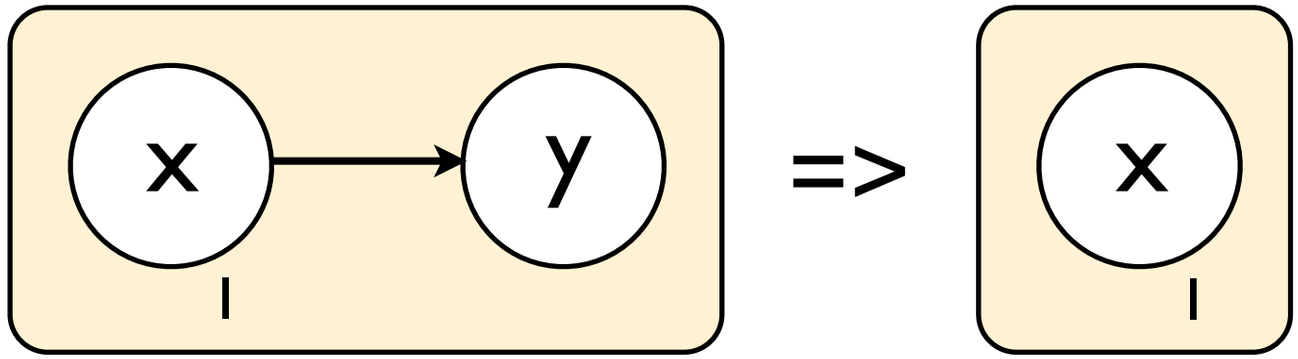
Deleting nodes

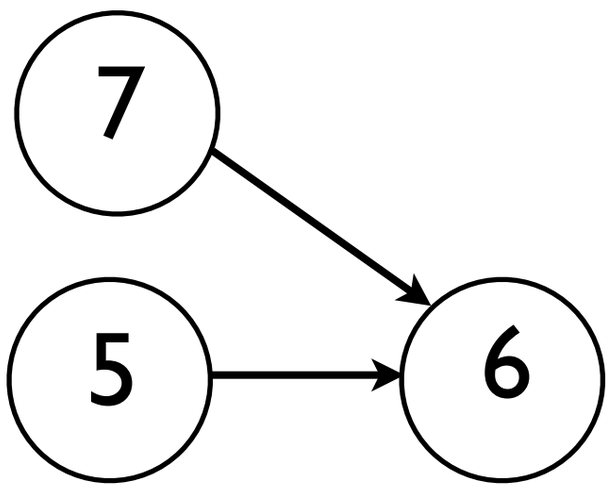
clip (x,y: int)



Deleting nodes

clip (x,y: int)

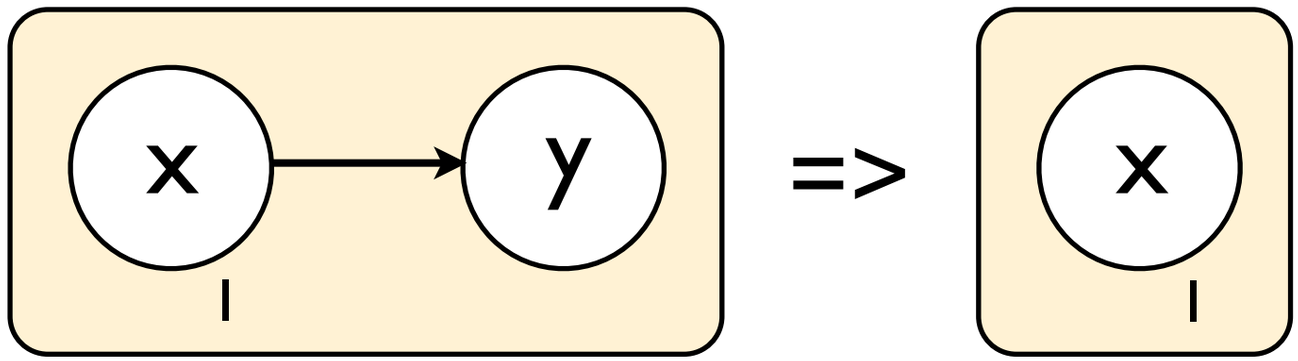


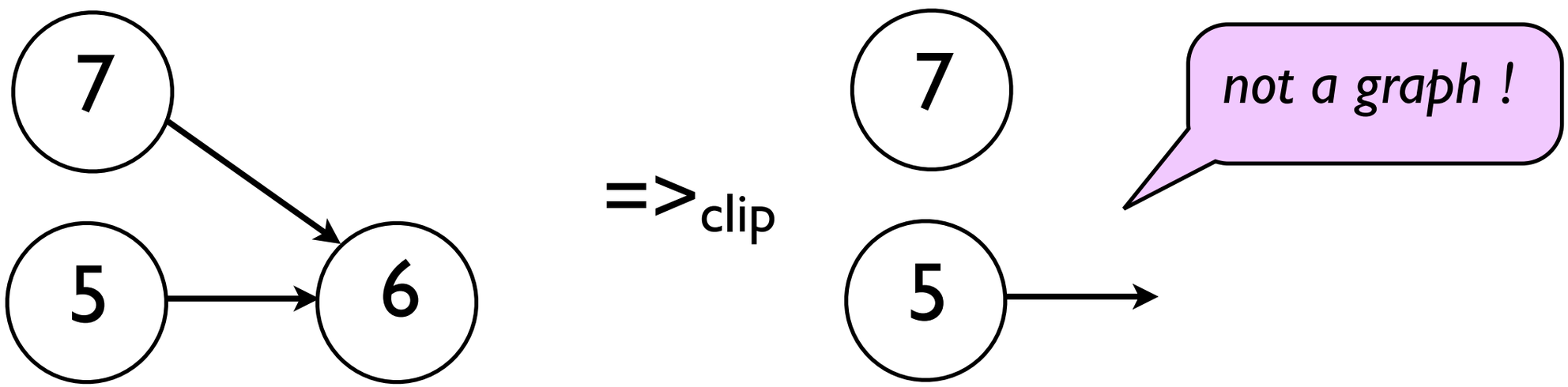


=>clip

Deleting nodes

clip (x,y: int)





Deleting nodes: a solution

- only allow rule applications that do not leave edges dangling

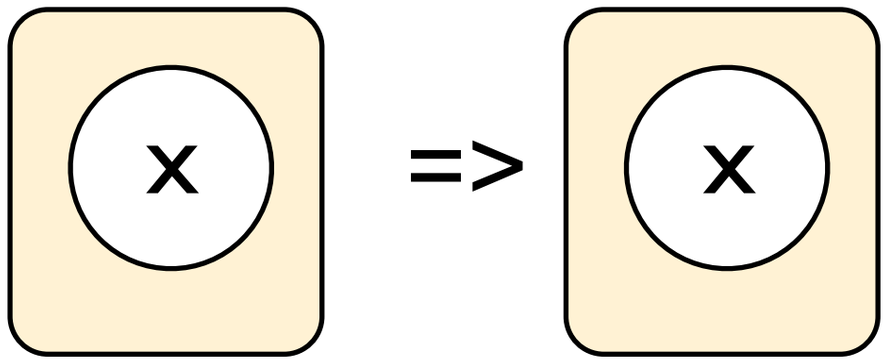
- satisfy the "dangling condition"

- called the double-pushout approach (DPO) to graph transformation

  - key property: rule applications are side-effect free

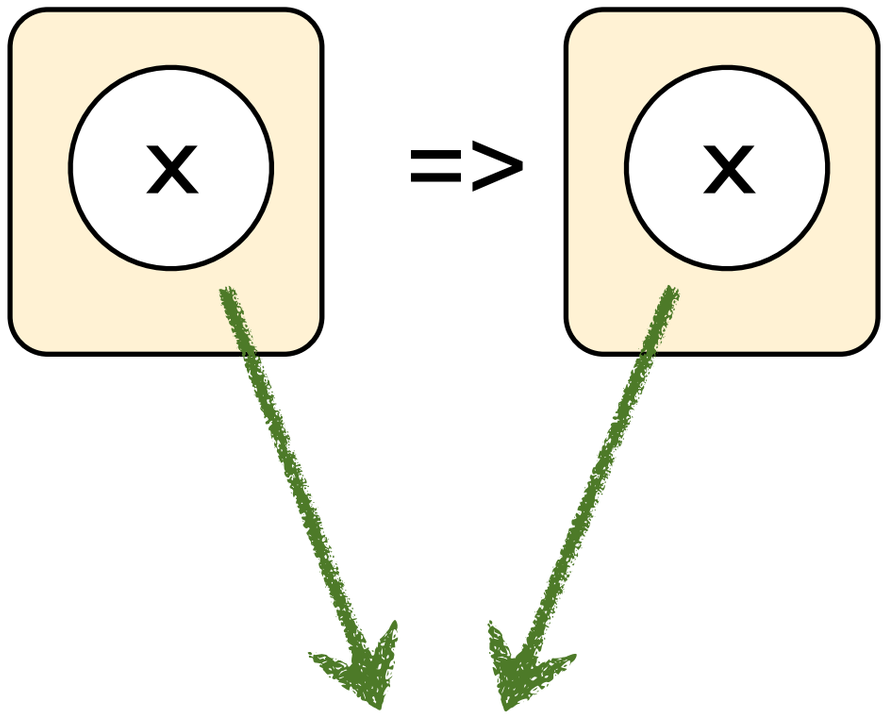
Deleting nodes: notation alert!

trickyRule (x: int)



Deleting nodes: notation alert!

trickyRule (x: int)

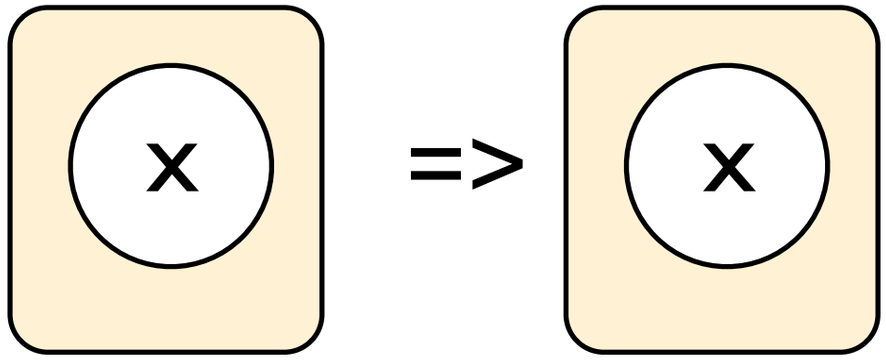


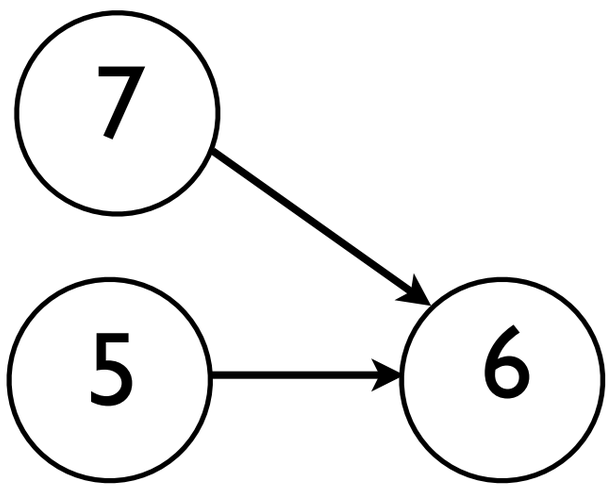
no number implies nodes are not the same

i.e. match of L^{α} is deleted, then recreated with the same label

No matches => failure

trickyRule (x: int)

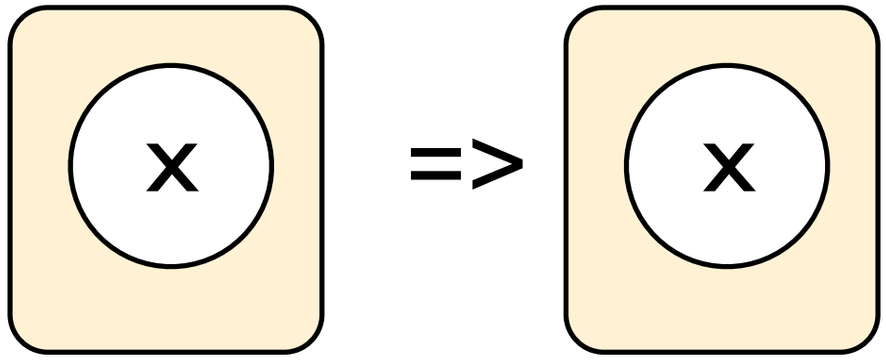


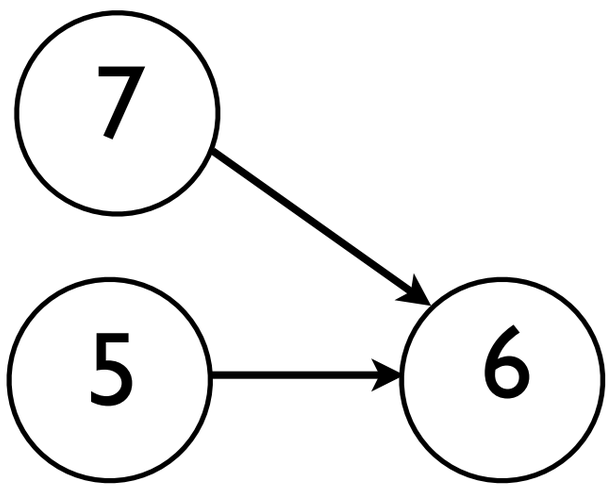


=>trickyRule

No matches => failure

trickyRule (x: int)



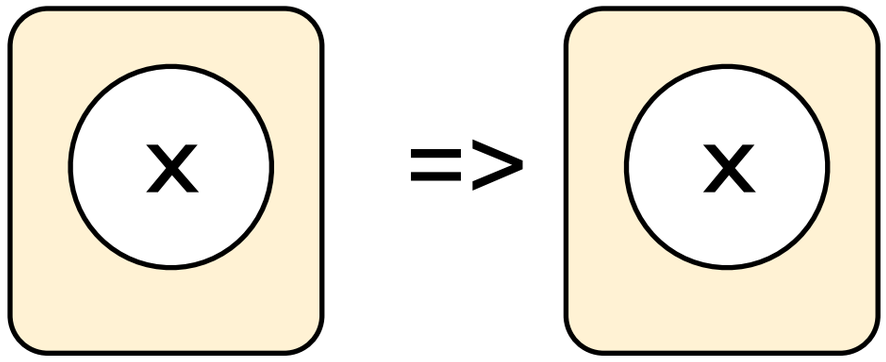


=>trickyRule

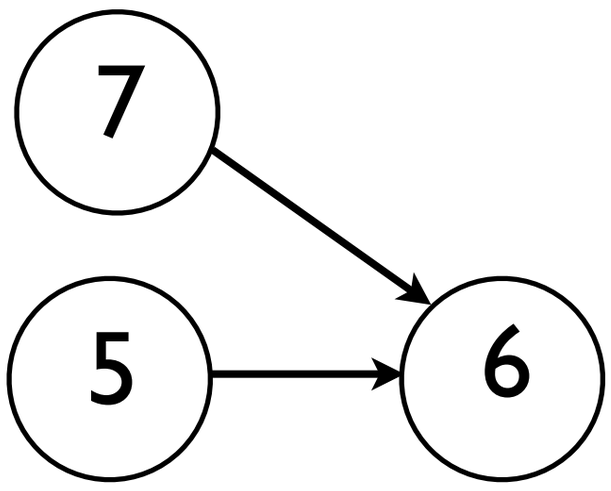
fail

No matches => failure

trickyRule (x: int)



no match since an edge would be left dangling
- program fails and terminates



=>$_{\text{trickyRule}}$ fail

Graph programs: control constructs

- simple core of control constructs

r	single rule application
$\{r_0, \ldots, r_n\}$	nondeterministic rule choice
$P; Q$	sequential composition
$\{r_0, \ldots, r_n\}!$	as-long-as-possible iteration

if $\{r_0, \ldots, r_n\}$ then P else Q

try $\{r_0, \ldots, r_n\}$ then P else Q

Graph programs: control constructs

- simple core of control constructs

r	single rule application
$\{r_0, \ldots, r_n\}$	nondeterministic rule choice
$P; Q$	sequential composition
$\{r_0, \ldots, r_n\}!$	as-long-as-possible iteration

if $\{r_0, \ldots, r_n\}$ then P else Q

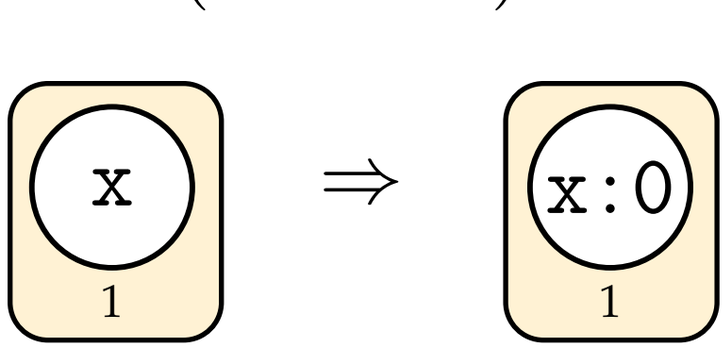
try $\{r_0, \ldots, r_n\}$ then P else Q

branch decided on whether guard fails or not
- in "if", the effects of guard not retained
- in "try", the effects of guard are retained

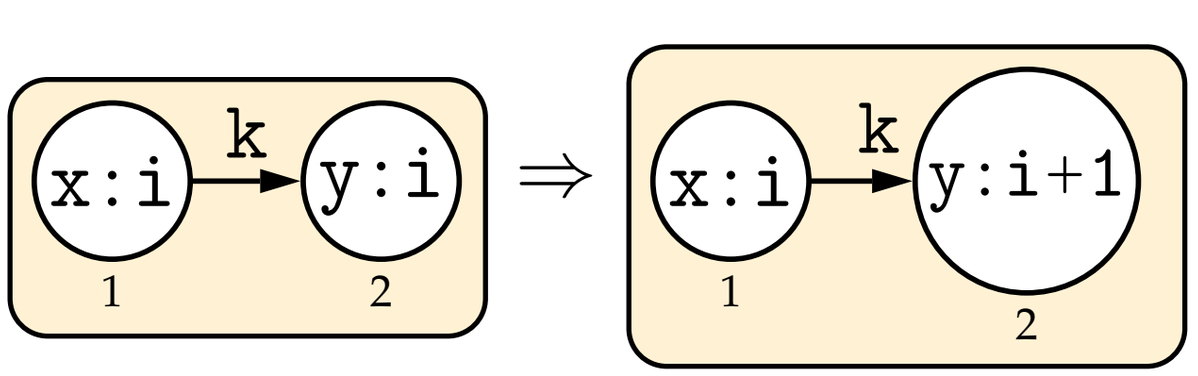
Example graph program: compute a colouring

colouring = init!; inc!

`init(x: atom)`



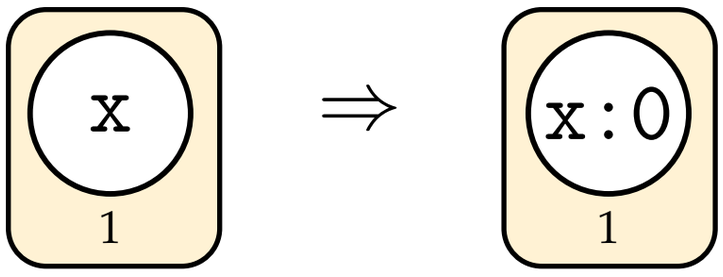
`inc(i: int; k: list; x, y: atom)`



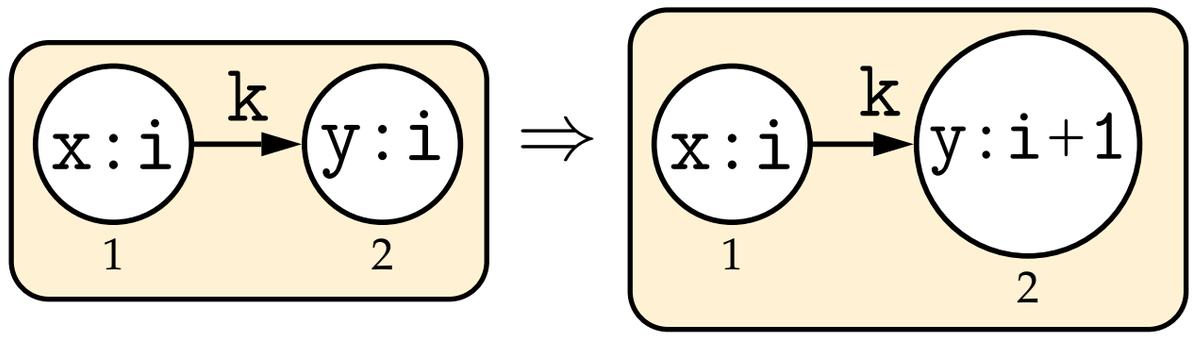
NB: type "atom" denotes an int or string; type "list" denotes an arbitrary sequence of ints/strings

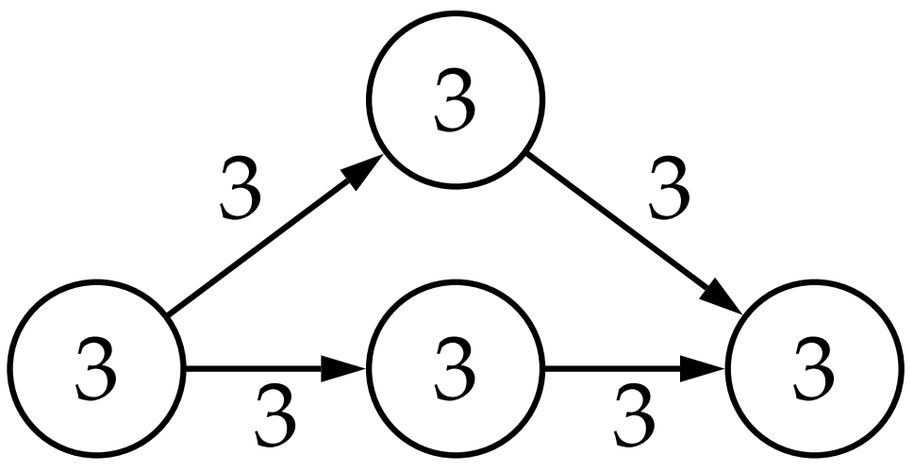
colouring = init!; inc!

`init(x: atom)`



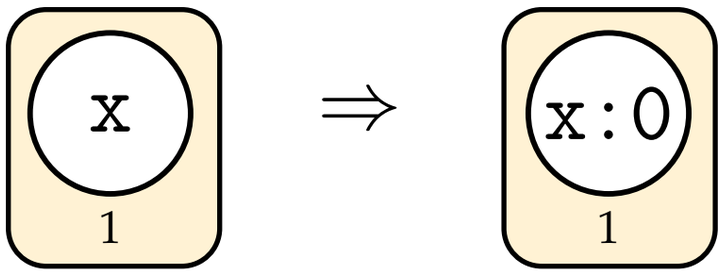
`inc(i: int; k: list; x, y: atom)`



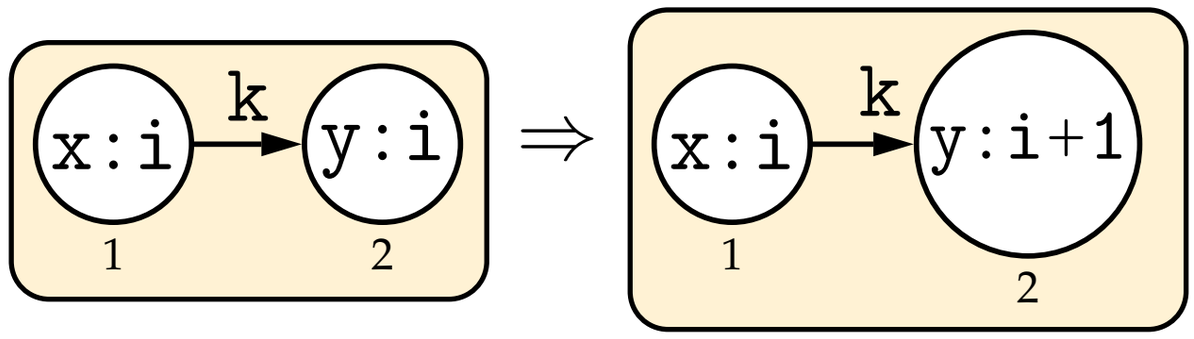


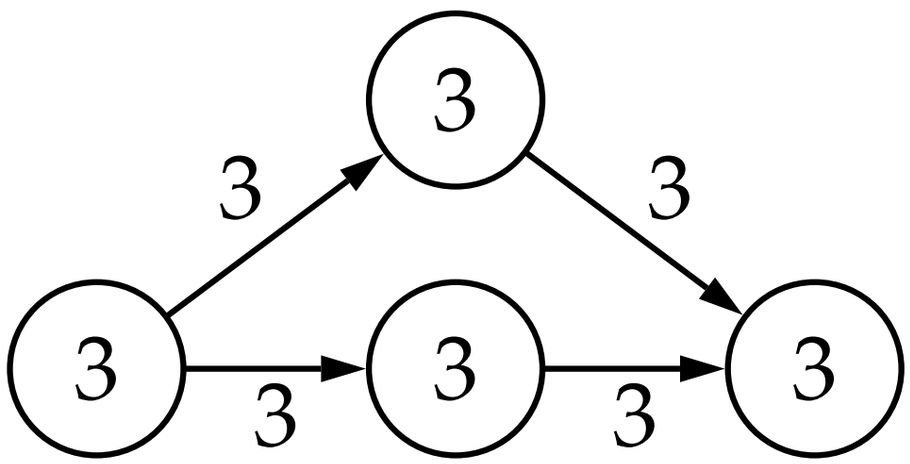
colouring = init!; inc!

`init(x: atom)`

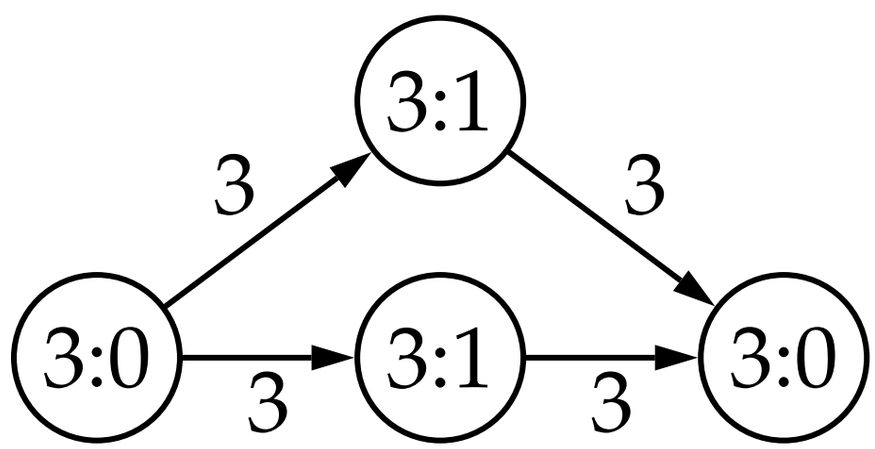


`inc(i: int; k: list; x, y: atom)`



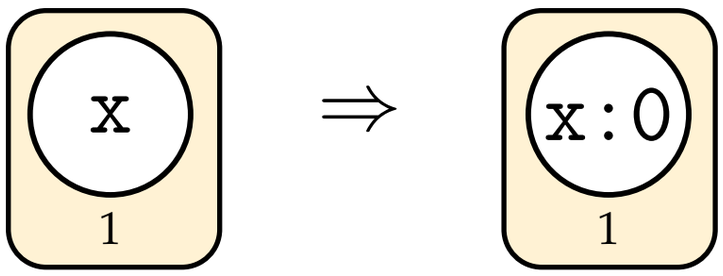


⇒+

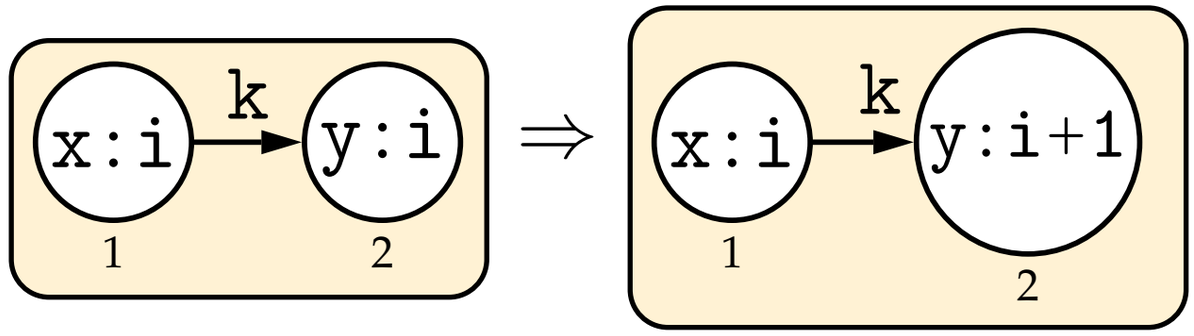


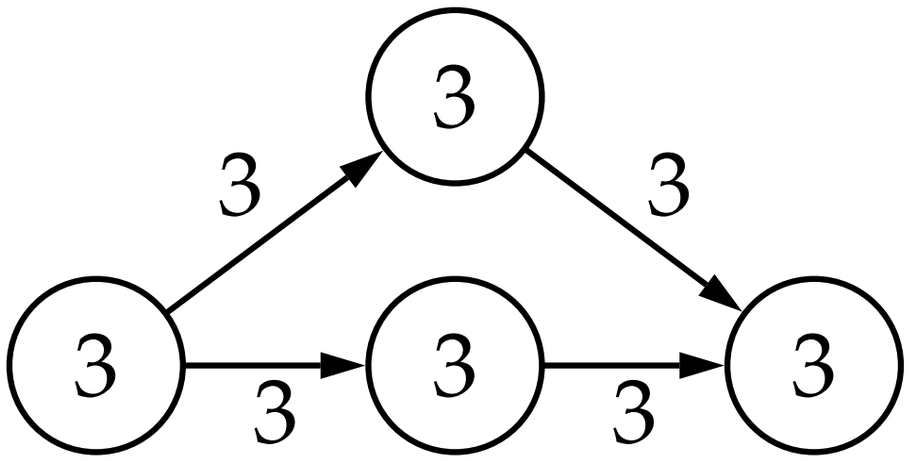
colouring = init!; inc!

`init(x: atom)`

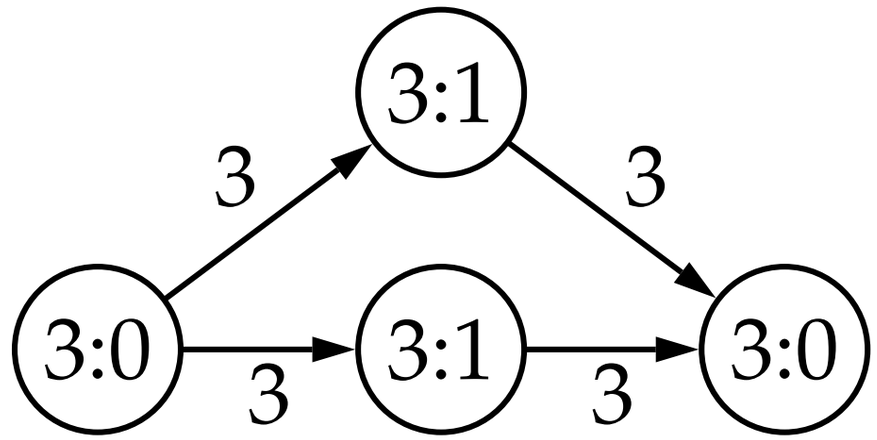


`inc(i: int; k: list; x, y: atom)`

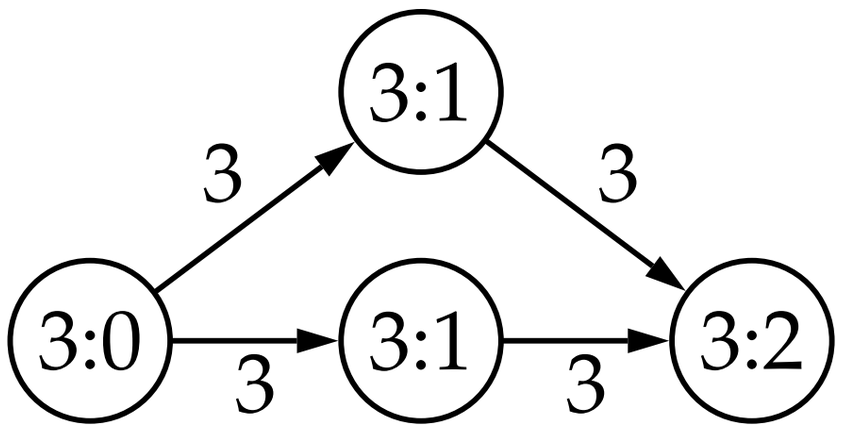




⇒+



⇒×



Next on the agenda

(1) a programming language for graphs 

(2) an assertion language for graphs

(3) Hoare-style reasoning about graph transformation

(4) program proofs

An assertion language for graphs

- we *could* use some first-order logic interpreted over graphs

$$\exists v.\ \mathrm{node}(v) \wedge l(v) = "party"$$

- but program states are graphs, and computational steps are graph transformation rules
  - both at a high-level of abstraction

- define an assertion language at the same level of abstraction
  - facilitate visual specifications
  - integrate the theory of graph transformation into assertional reasoning

E-conditions: graphical assertions about the program state

- E-conditions - a logical assertion language embedding pictures of graphs

- *idea:* express the existence of some subgraph with particular structural features and particular relations between labels

- combine with Boolean negation and connectives for a visual logic equivalent to first-order logic over graphs

- *historical note:* a generalisation of nested conditions (Habel & Pennemann)

E-conditions: graphical assertions about the program state

true

$\exists(C \mid \gamma, c')$

E-conditions: graphical assertions about the program state

true

$\exists(C \mid \gamma, c')$

a constant symbol

E-conditions: graphical assertions about the program state

true

$\exists(C \mid \gamma, c')$

C is a graph labelled over expressions

E-conditions: graphical assertions about the program state

true

$\exists(C \mid \gamma, c')$

γ is a constraint over the labels of C

E-conditions: graphical assertions about the program state

true

$\exists(C \mid \gamma, c')$

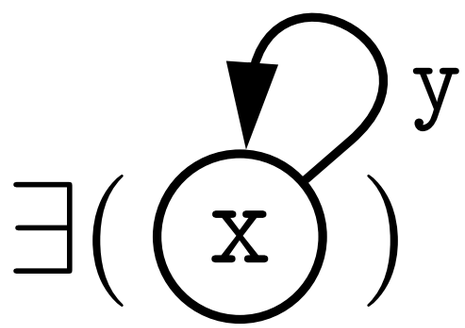
c' is an E-condition, i.e. **true** *or* $\exists(C' \mid \gamma', c'')$

Semantics by example

true

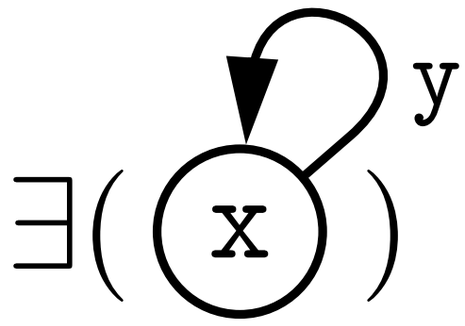
satisfied by all graphs

Semantics by example



“there exists a node incident to a loop”

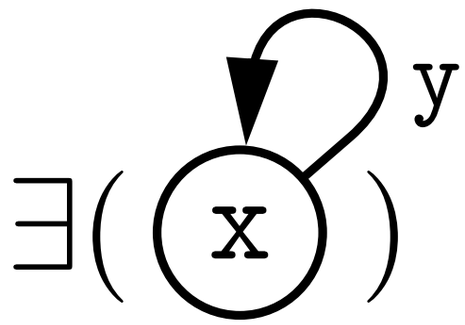
Semantics by example



variables are untyped (and here, unconstrained)

"there exists a node incident to a loop"

Semantics by example

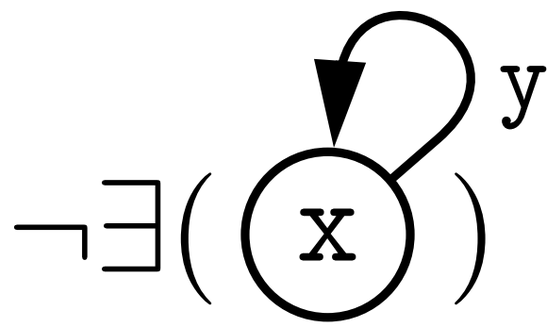


$\exists(C \mid \gamma, c')$

we don't write γ *or c' if they are just* true

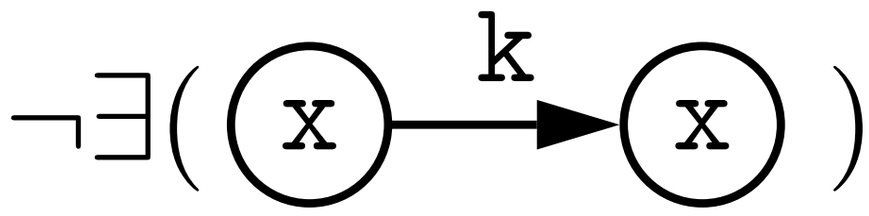
"there exists a node incident to a loop"

Boolean expressions over E-conditions are also E-conditions

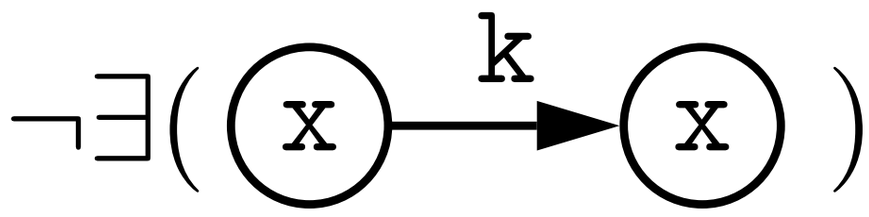


"there does <u>not</u> exist a node incident to a loop"

Expressions as labels are important, too!

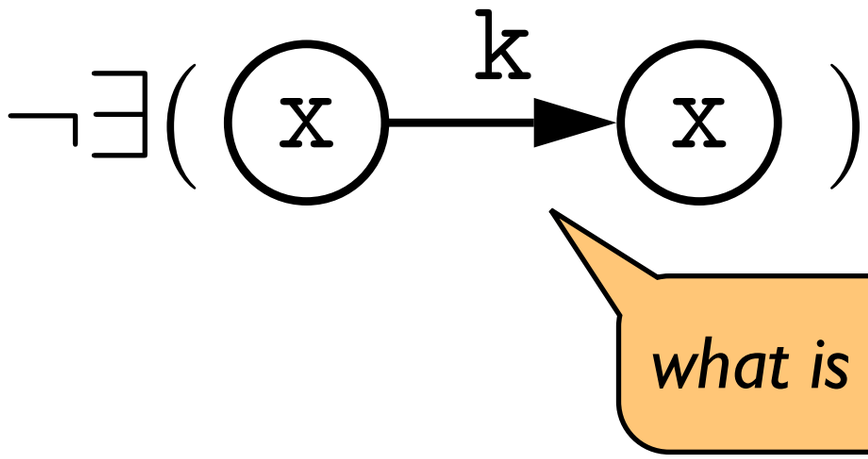


Expressions as labels are important, too!



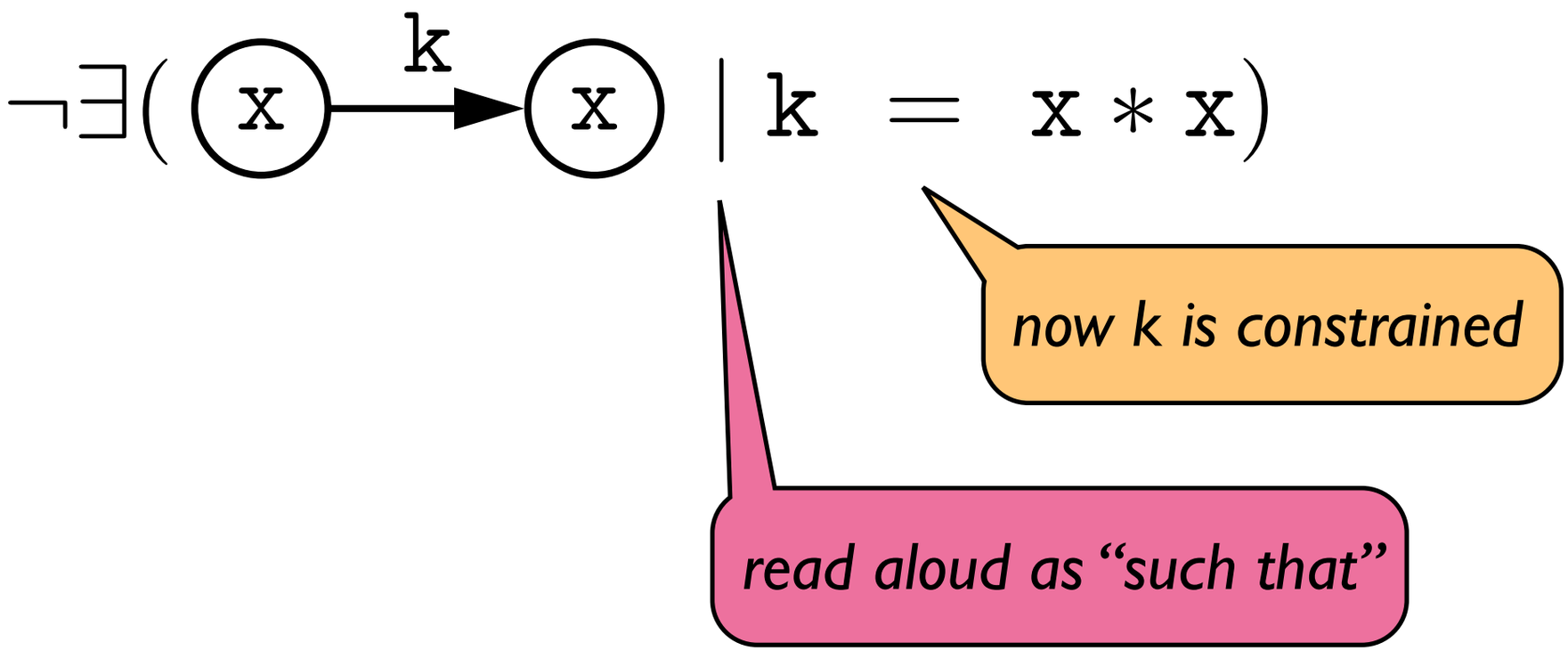
"there does <u>not</u> exist a pair of adjacent nodes with the same label"

Expressions as labels are important, too!

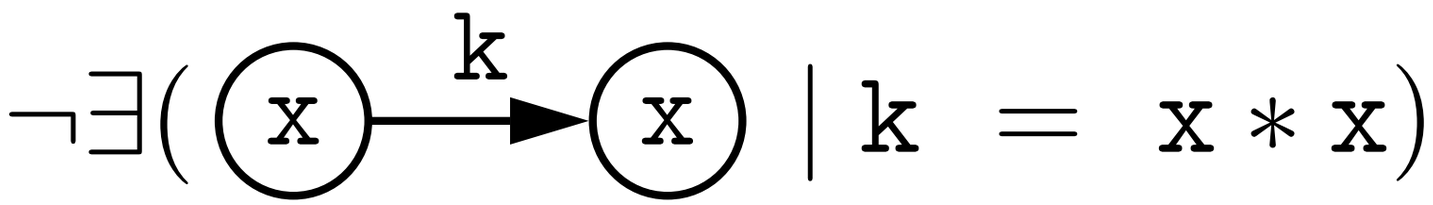


"there does <u>not</u> exist a pair of adjacent nodes with the same label"

Expressions as labels are important, too!



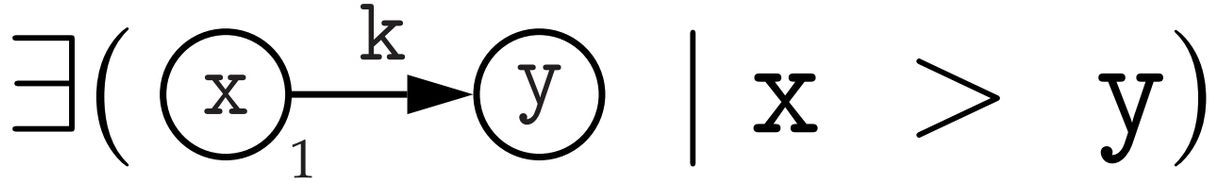
Expressions as labels are important, too!



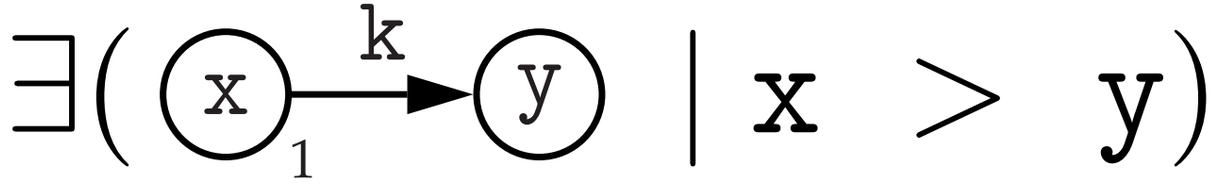
now k is constrained

"there does <u>not</u> exist a pair of adjacent nodes such that:
(1) the nodes have the same label; and
(2) the edge is the square of that label"

Constraints enforce relations between labels

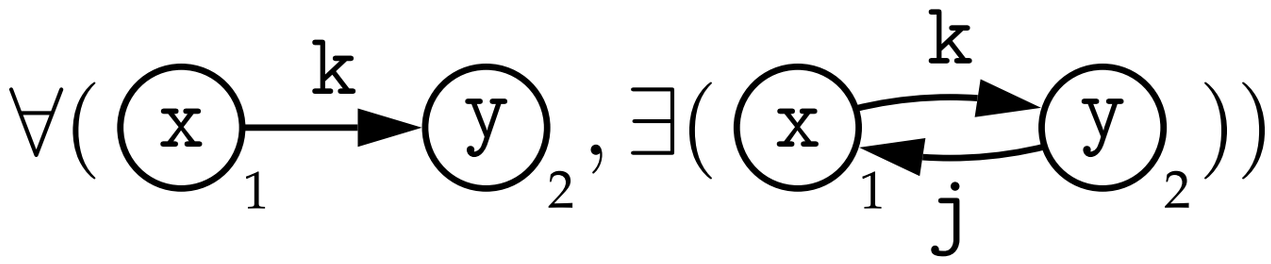


Constraints enforce relations between labels

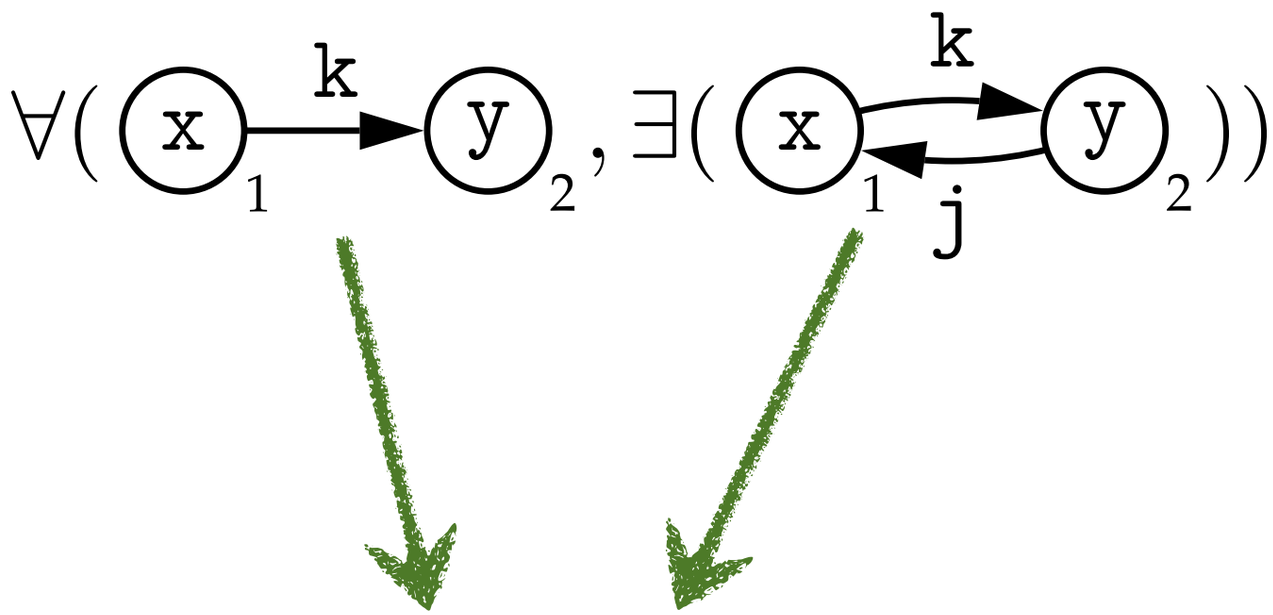


"there is a pair of adjacent nodes with source label greater than target label AND no node is labelled with a negative number"

Nesting allows assertions about specific contexts



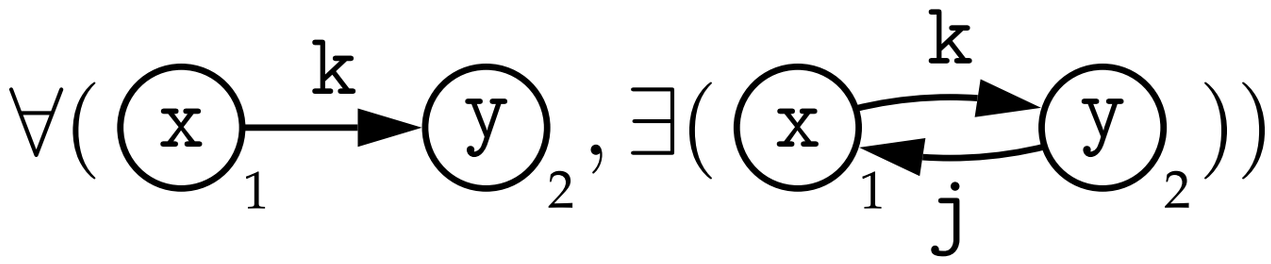
Nesting allows assertions about specific contexts



these nodes and the k*-labelled edge are the same*

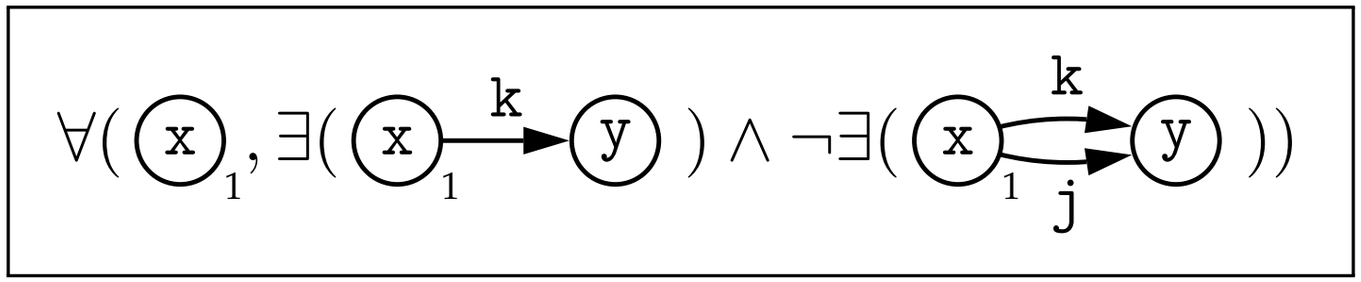
- *nesting allows us to express properties about particular contexts*
  i.e. express something about the universally quantified graph

Nesting allows assertions about specific contexts

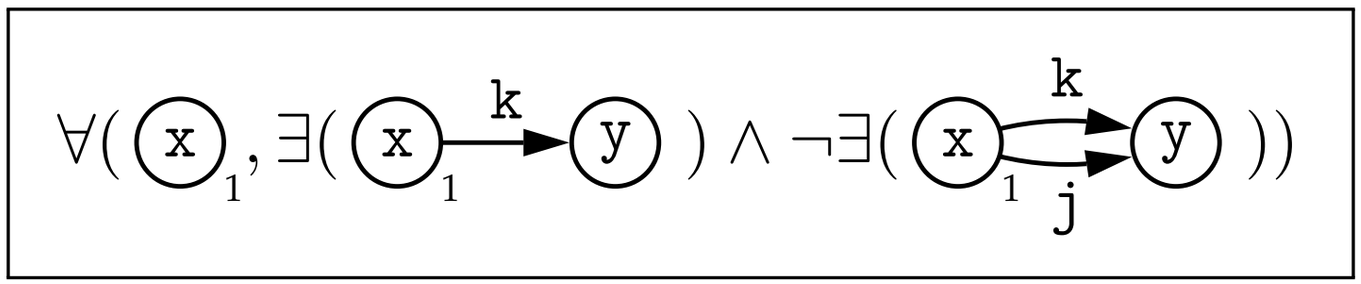


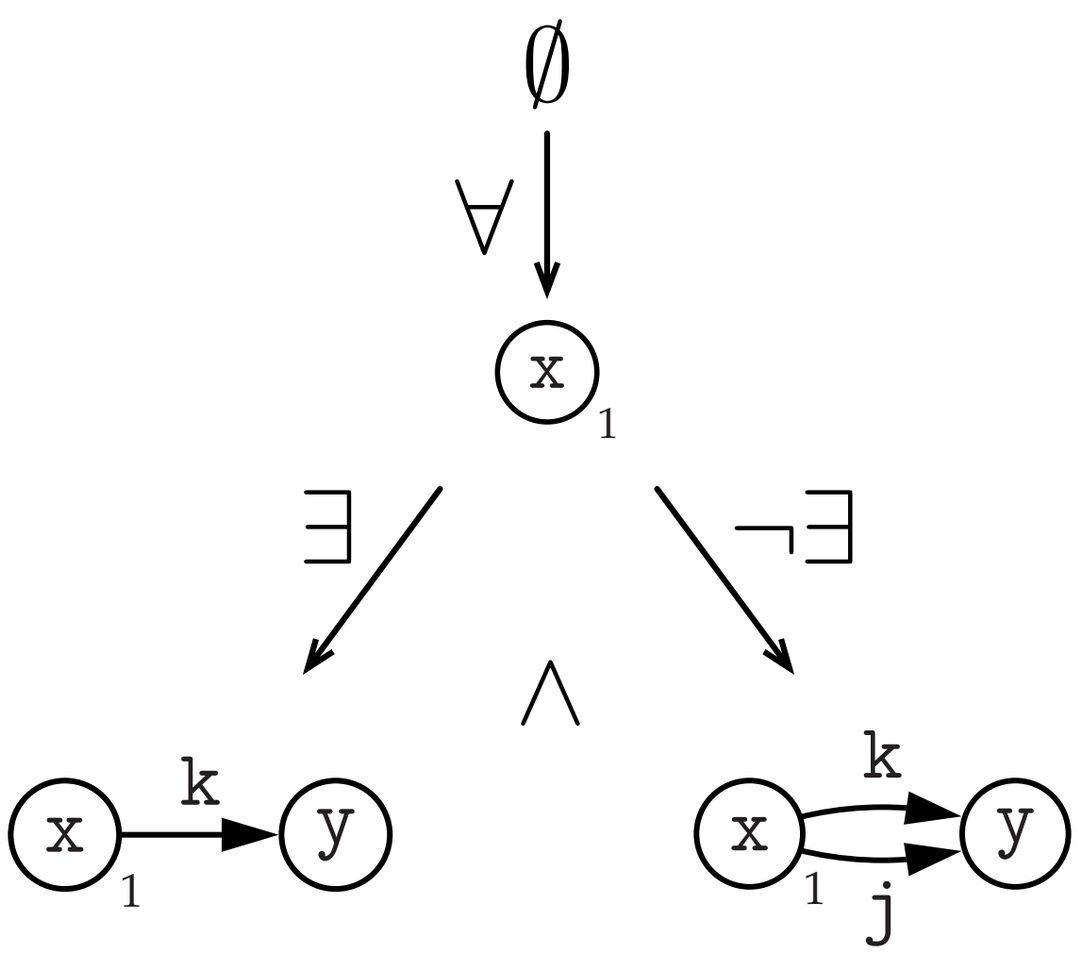
"if there is an edge from v *to* w, *then there is also an edge from* w *to* v (graph is undirected)"

View as a “tree” that is building up structural and relational information

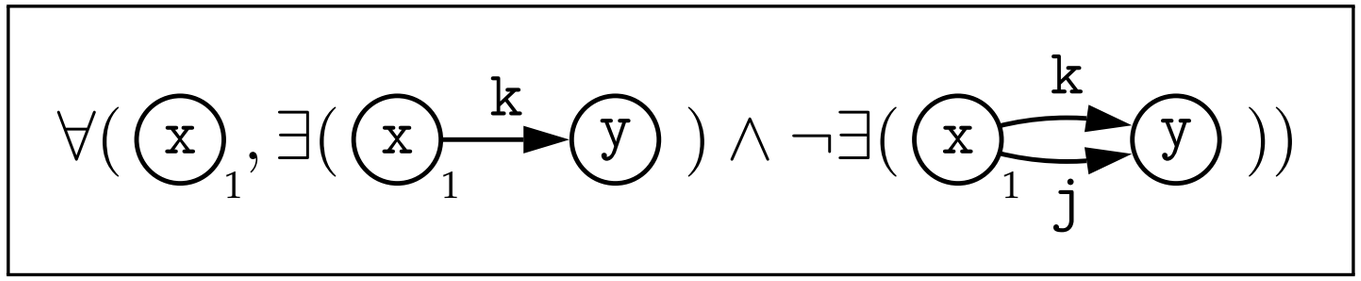


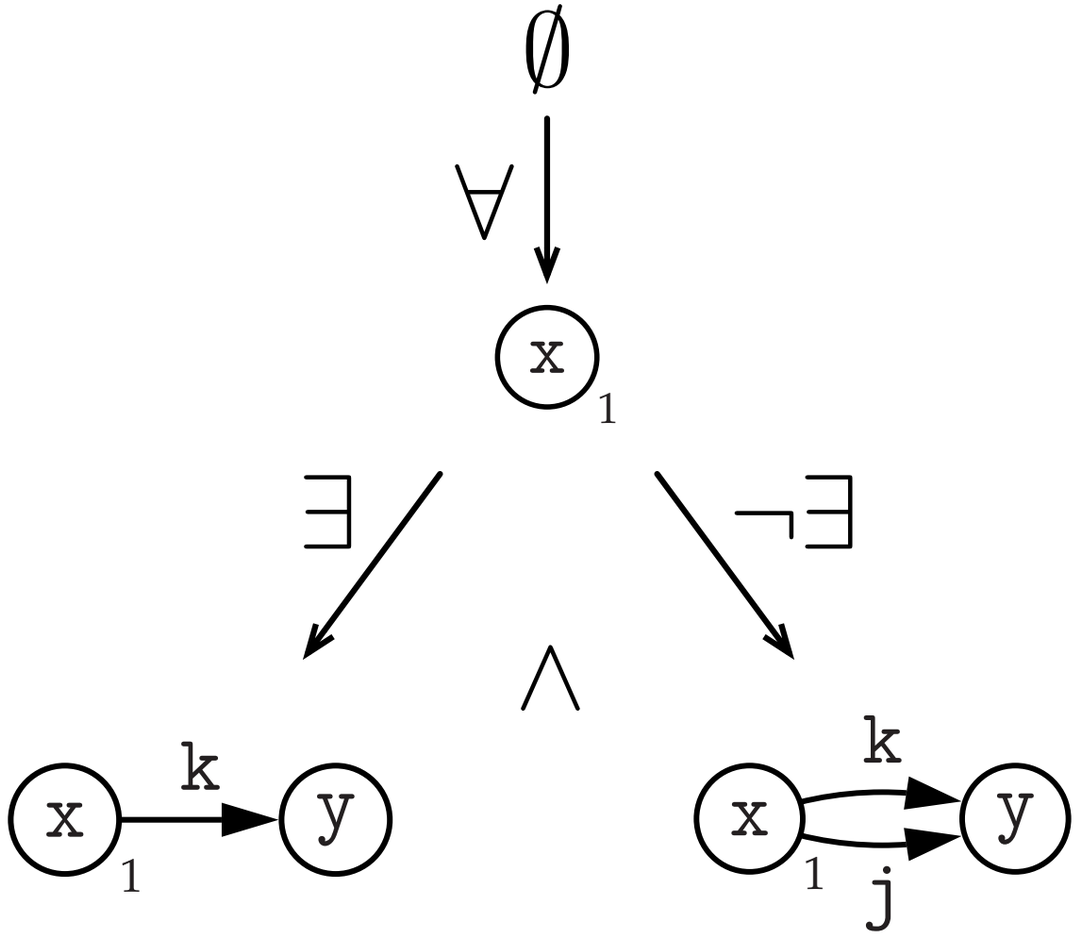
View as a "tree" that is building up structural and relational information





View as a "tree" that is building up structural and relational information





for every node,
(1) there exists an outgoing edge to another node; and
(2) there does not exist a pair of outgoing edges to another node

Satisfaction of E-conditions
(an incomplete definition)

- a graph *G* satisfies an E-condition *c*, written $G \models c$, if:

  (1) c = true; or

Satisfaction of E-conditions
(an incomplete definition)

- a graph *G* satisfies an E-condition *c*, written $G \models c$, if:

  (1) c = true; or

  (2) c = $\exists(C \mid \gamma, c')$

  and there is a mapping α:Vars -> Data such that

  (a) [|γ|]α = true;
  (b) C^{α} is a subgraph of *G*;
  (c) the context of C^{α} in *G* "satisfies" *c'*

Satisfaction of E-conditions
(an incomplete definition)

- a graph *G* satisfies an E-condition *c*, written $G \models c$, if:

  (1) c = true; or

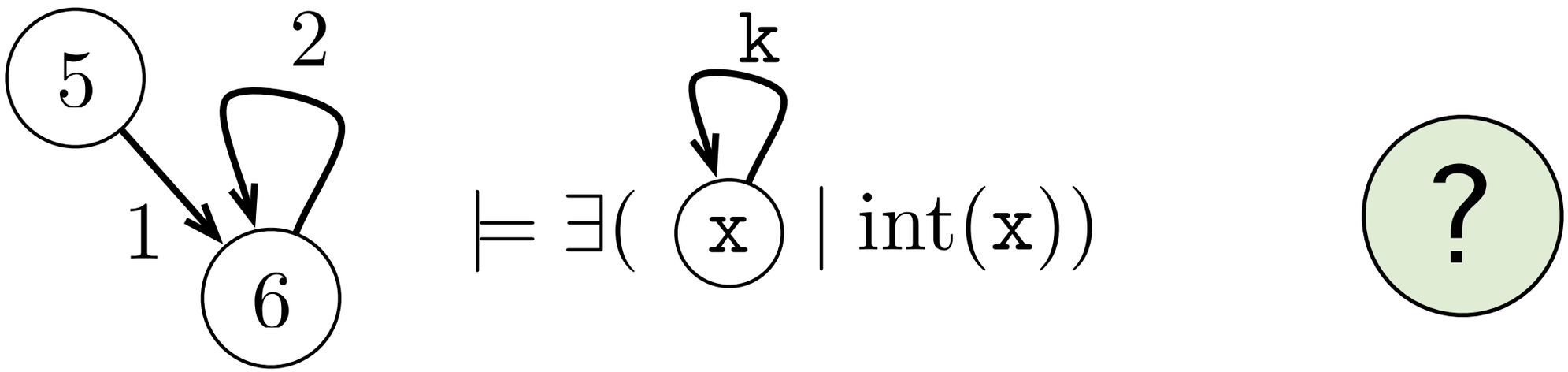
  (2) c = $\exists(C \mid \gamma, c')$

  and there is a mapping α:Vars -> Data such that

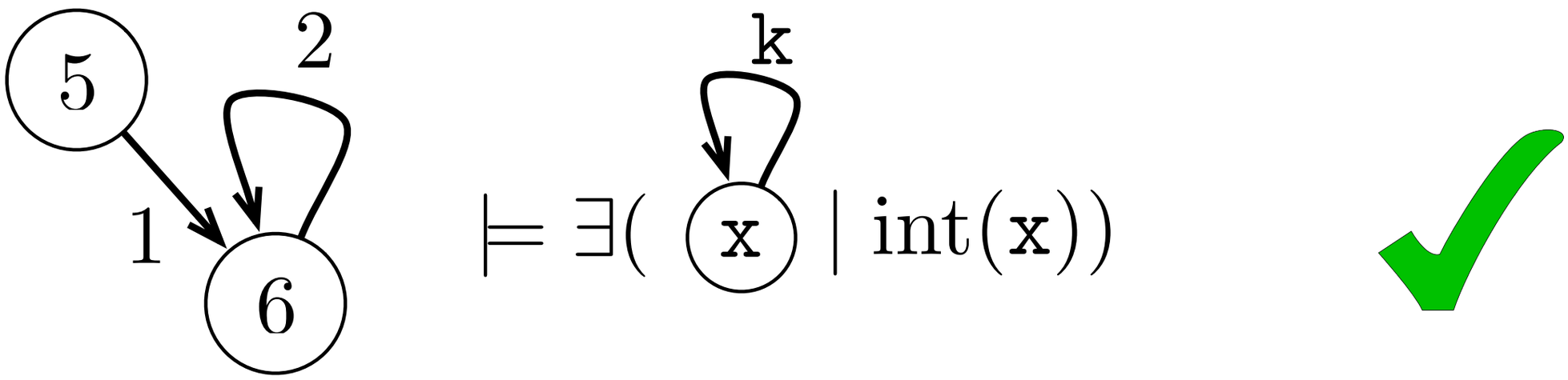
  (a) [|γ|]α = true;
  (b) C^{α} is a subgraph of *G*;
  (c) the context of C^{α} in *G* "satisfies" *c'*

the complete formal definition needs the notion of "morphism" for the inductive part (c)

Satisfaction by example



Satisfaction by example



by assignment x |-> 6, k |-> 2

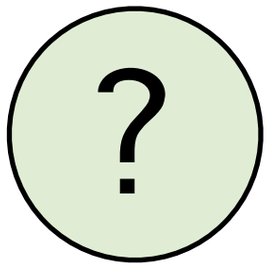
What can E-conditions not specify?

- E-conditions are expressively equivalent to a first-order logic interpreted over graphs

- for relations between labels, this is great...

- ...but for graphs, first-order logic is quite weak for expressing structure

  - only "local" properties
  - need more than FO for path properties, connectedness...



what prevents us from simply adding predicates for these properties?

Next on the agenda

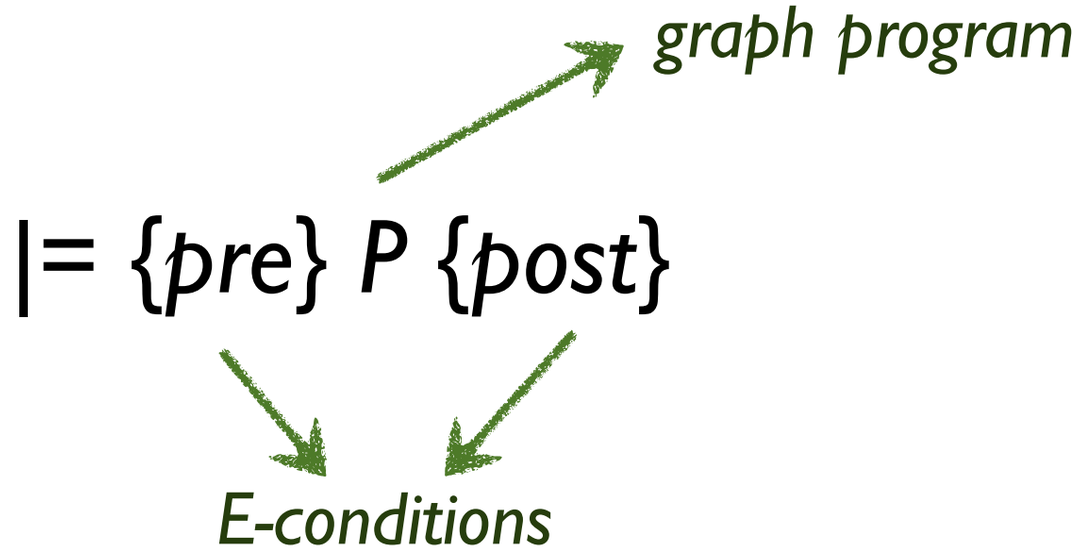
(1) a programming language for graphs 

(2) an assertion language for graphs 

(3) Hoare-style reasoning about graph transformation

(4) program proofs

Partial correctness specifications



- partial correctness: if program *P* is executed on a graph *G* such that *G* |= *pre*, then if a graph *H* results, *H* |= *post*

- differs to the partial correctness definition in lecture 2:

  - nondeterminism: many graphs *H* could result, but all guaranteed to satisfy *post*
  - *P* might fail on *G*

Proof rules

$$\text{[ruleset]}\ \frac{\{c\}\ r\ \{d\}\ \text{for each}\ r \in \mathcal{R}}{\{c\}\ \mathcal{R}\ \{d\}}$$

where: $\mathcal{R} = \{r_0, \ldots, r_n\}$

Proof rules

$$\text{[comp]}\ \frac{\{c\}\ P\ \{e\} \quad \{e\}\ Q\ \{d\}}{\{c\}\ P; Q\ \{d\}}$$

Proof rules

$$[!]\ \frac{\{inv\}\ \mathcal{R}\ \{inv\}}{\{inv\}\ \mathcal{R}!\ \{inv \wedge \neg \mathrm{App}(\mathcal{R})\}}$$

where App(R) constructs an E-condition expressing that R will not fail on the graph

Proof rules

$$\text{[if]}\ \frac{\{c \wedge \mathrm{App}(\mathcal{R})\}\ P\ \{d\} \quad \{c \wedge \neg\mathrm{App}(\mathcal{R})\}\ Q\ \{d\}}{\{c\}\ \texttt{if}\ \mathcal{R}\ \texttt{then}\ P\ \texttt{else}\ Q\ \{d\}}$$

$$\text{[try]}\ \frac{\{c \wedge \mathrm{App}(\mathcal{R})\}\ \mathcal{R};\ P\ \{d\} \quad \{c \wedge \neg\mathrm{App}(\mathcal{R})\}\ Q\ \{d\}}{\{c\}\ \texttt{try}\ \mathcal{R}\ \texttt{then}\ P\ \texttt{else}\ Q\ \{d\}}$$

Proof rules

$$[\text{cons}]\ \frac{c \Rightarrow c' \quad \{c'\}\ P\ \{d'\} \quad d' \Rightarrow d}{\{c\}\ P\ \{d\}}$$

how to show the validity of an E-condition?

Axioms

$$[\text{nonapp}]\ \frac{}{\{\neg \mathrm{App}(\{r\})\}\ r\ \{\texttt{false}\}}$$

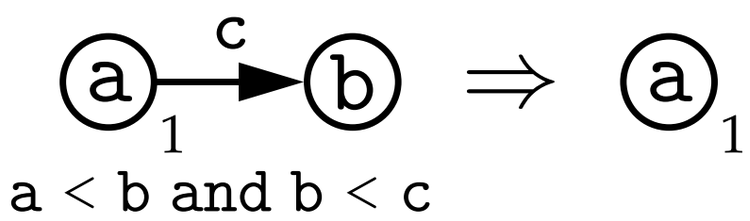
Axioms

$$[\text{ruleapp}]\ \frac{}{\{\text{Pre}(r, c)\}\ r\ \{c\}}$$

where Pre(*r*,*c*) *constructs an E-condition expressing the weakest precondition that must hold for r to establish c*

What does App(R) look like?
(see Poskitt 13 for the construction)

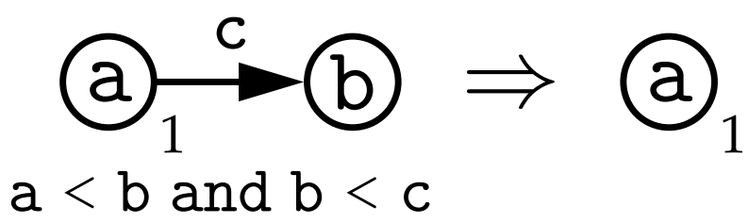
reduce(a,b,c: int)



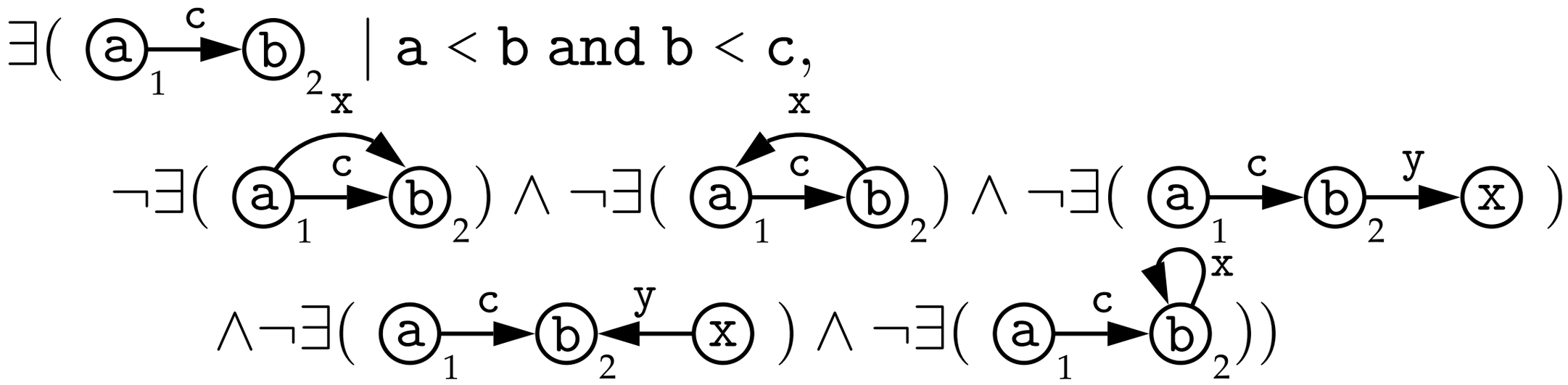
an E-condition equivalent to the following is constructed by App(reduce):

What does App(R) look like?
(see Poskitt 13 for the construction)

reduce(a,b,c: int)

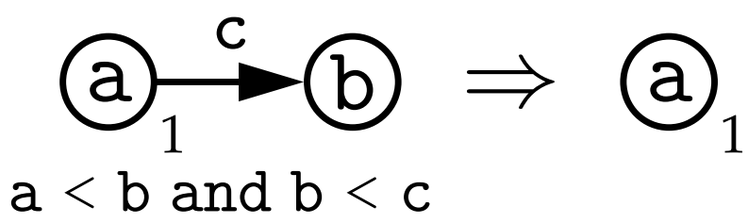


an E-condition equivalent to the following is constructed by App(reduce):



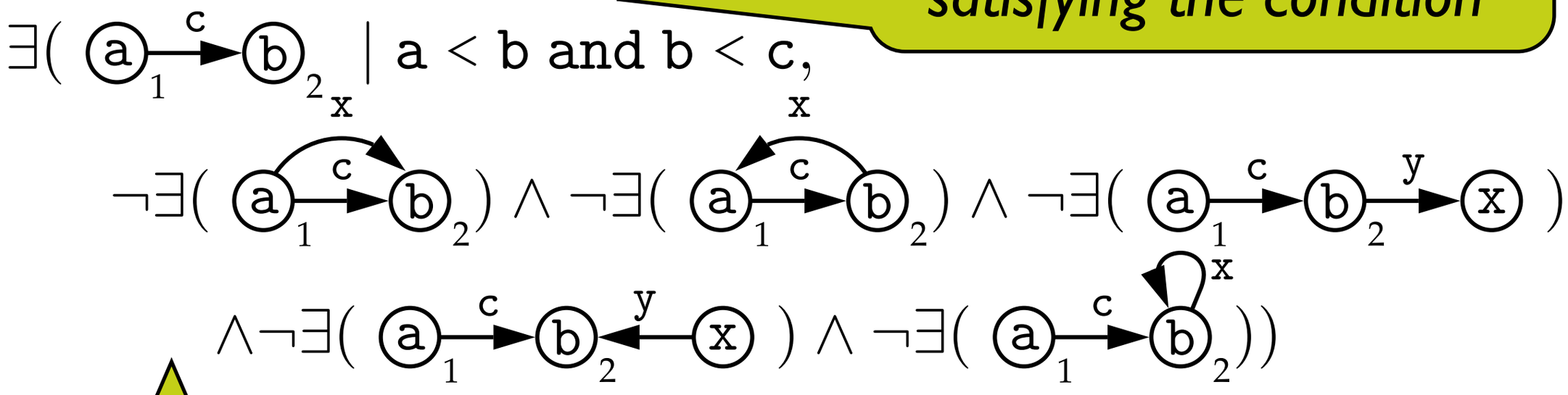
What does App(R) look like?
(see Poskitt 13 for the construction)

reduce(a,b,c: int)



an E-condition equivalent to the following is constructed by App(reduce):

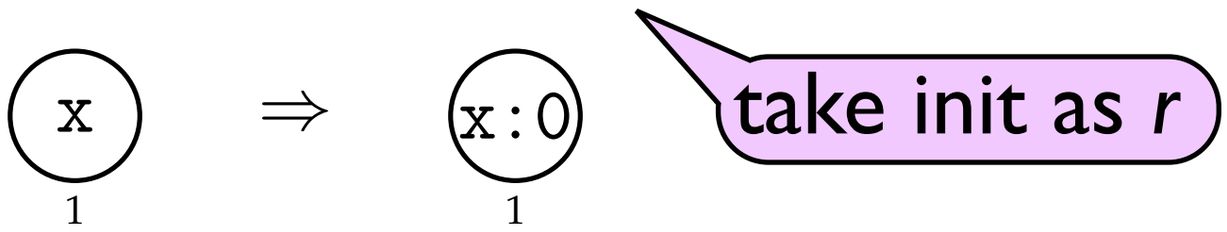
there is a potential match satisfying the condition



context satisfies dangling condition!

What does Pre(r,c) look like?
(see Poskitt 13 for the construction)

`init(x: atom)`



$\forall(\ @_1, \exists(\ @_1 \mid \texttt{atom(a)}\ ) \vee \exists(\ @_1 \mid \texttt{a} = \texttt{b:c and atom(b) and c} >= 0))$

take as postcondition c
"every node is either labelled by (1) an atom; or (2) a sequence b:c with b an atom, c a natural

What does Pre(r,c) look like?

(see Poskitt 13 for the construction)

take home point:
embeds the left-hand graph of *r* and the postcondition *c* together in a precondition

$$\forall(\ \textcircled{x}_1 \mid \texttt{atom(x)},$$

$$\forall(\ \textcircled{x}_1\ \textcircled{a}_2, \exists(\ \textcircled{x}_1\ \textcircled{a}_2 \mid \texttt{atom(a)}\ )$$

$$\vee\ \exists(\ \textcircled{x}_1\ \textcircled{a}_2 \mid \texttt{a} = \texttt{b:c and atom(b) and c} >= 0))$$

$$\wedge\ \forall(\ \textcircled{x}_1, \exists(\ \textcircled{x}_1 \mid \texttt{atom(x:0)}\ )$$

$$\vee\ \exists(\ \textcircled{x}_1 \mid \texttt{x:0} = \texttt{b:c and atom(b) and c} >= 0)))$$

Instance of [ruleapp] axiom for init, c

$$\{\ \forall(\ \textcircled{x}_1 \mid \texttt{atom(x)},\ \forall(\ \textcircled{x}_1\ \textcircled{a}_2, \exists(\ \textcircled{x}_1\ \textcircled{a}_2 \mid \texttt{atom(a)})\ \vee \exists(\ \textcircled{x}_1\ \textcircled{a}_2 \mid \texttt{a} = \texttt{b:c and atom(b) and c} >= 0))\ \wedge \forall(\ \textcircled{x}_1, \exists(\ \textcircled{x}_1 \mid \texttt{atom(x:0)})\ \vee \exists(\ \textcircled{x}_1 \mid \texttt{x:0} = \texttt{b:c and atom(b) and c} >= 0)))\ \}$$

init

$$\{\ \forall(\ \textcircled{a}_1, \exists(\ \textcircled{a}_1 \mid \texttt{atom(a)})\ \vee \exists(\ \textcircled{a}_1 \mid \texttt{a} = \texttt{b:c and atom(b) and c} >= 0))\ \}$$

Classic Hoare logic vs. graph program Hoare logic

- for the most part, classic and graph-based Hoare logic are very similar

- but interestingly, in "raising the abstraction" of programs and assertions, we make the core axiom of the Hoare logic very technical / complicated

- compare to the simplicity of the assignment axiom

- motivates tool support, especially for generating App(R), Pre(r,c), and for deciding implications that have them as consequences

Next on the agenda

(1) a programming language for graphs 

(2) an assertion language for graphs 

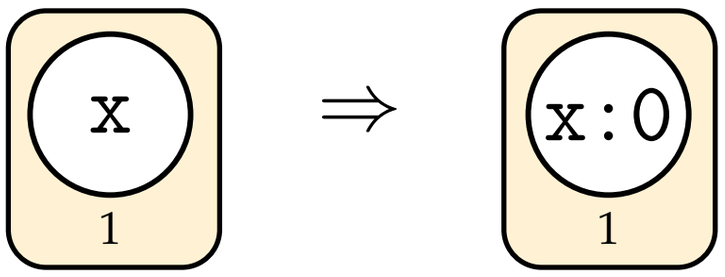
(3) Hoare-style reasoning about graph transformation 

(4) program proofs

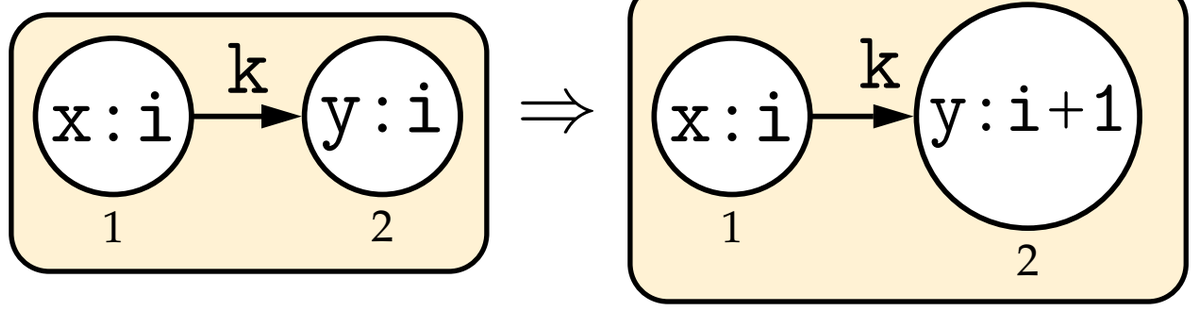
Partial correctness of colouring

colouring = init!; inc!

`init(x: atom)`



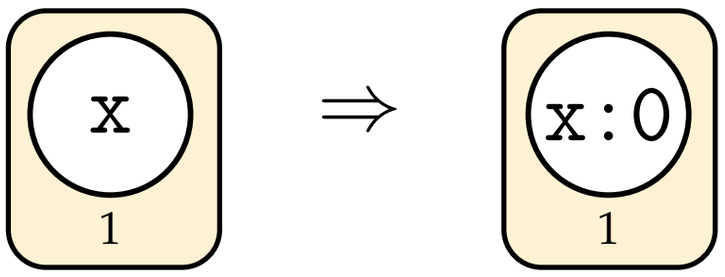
`inc(i: int; k: list; x, y: atom)`



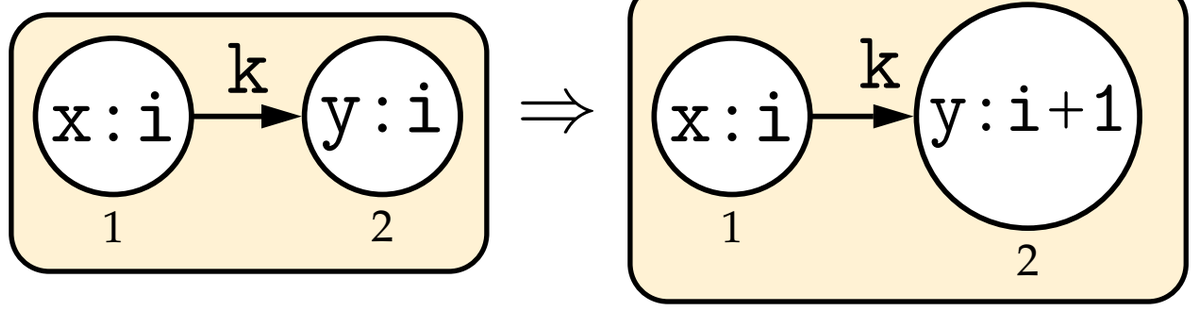
Partial correctness of colouring

colouring = init!; inc!

`init(x: atom)`



`inc(i: int; k: list; x, y: atom)`



$$\{\ \forall(\ @_1, \exists(\ @_1 \mid \texttt{atom(a)}))\ \}$$

init!; inc!

$$\left\{\begin{array}{l} \forall(\ @_1, \exists(\ @_1 \mid \texttt{a} = \texttt{b:c and atom(b) and c} >= 0)) \\ \wedge\ \neg\exists(\ \texttt{x:i} \xrightarrow{\texttt{k}} \texttt{y:i} \mid \texttt{atom(x,y) and int(i)}) \end{array}\right\}$$

!!!

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\{\mathrm{Pre}(\texttt{init}, e)\}\ \texttt{init}\ \{e\}}[\text{ruleapp}]}{\{e\}\ \texttt{init}\ \{e\}}[\text{cons}]}{\{e\}\ \texttt{init!}\ \{e \wedge \neg \mathrm{App}(\{\texttt{init}\})\}}[!]}{\{c\}\ \texttt{init!}\ \{d\}}[\text{cons}] \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{\{\mathrm{Pre}(\texttt{inc}, d)\}\ \texttt{inc}\ \{d\}}[\text{ruleapp}]}{\{d\}\ \texttt{inc}\ \{d\}}[\text{cons}]}{\{d\}\ \texttt{inc!}\ \{d \wedge \neg \mathrm{App}(\{\texttt{inc}\})\}}[!]}{\vdash_{\text{par}} \{c\}\ \texttt{init!};\ \texttt{inc!}\ \{d \wedge \neg \mathrm{App}(\{\texttt{inc}\})\}}[\text{comp}]$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\{\mathrm{Pre}(\texttt{init}, e)\}\ \texttt{init}\ \{e\}}\,[\text{ruleapp}]}{\{e\}\ \texttt{init}\ \{e\}}\,[\text{cons}]}{\{e\}\ \texttt{init!}\ \{e \wedge \neg \mathrm{App}(\{\texttt{init}\})\}}\,[!]}{\{c\}\ \texttt{init!}\ \{d\}}\,[\text{cons}] \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{\{\mathrm{Pre}(\texttt{inc}, d)\}\ \texttt{inc}\ \{d\}}\,[\text{ruleapp}]}{\{d\}\ \texttt{inc}\ \{d\}}\,[\text{cons}]}{\{d\}\ \texttt{inc!}\ \{d \wedge \neg \mathrm{App}(\{\texttt{inc}\})\}}\,[!]}{\vdash_{\text{par}} \{c\}\ \texttt{init!};\ \texttt{inc!}\ \{d \wedge \neg \mathrm{App}(\{\texttt{inc}\})\}}\,[\text{comp}]$$

$$c \quad = \quad \forall(\ \textcircled{a}_1, \exists(\ \textcircled{a}_1 \mid \texttt{atom(a)}\ ))$$

$$d \quad = \quad \forall(\ \textcircled{a}_1, \exists(\ \textcircled{a}_1 \mid \texttt{a} = \texttt{b:c and atom(b) and c} >= 0))$$

$$\neg\mathrm{App}(\{\texttt{inc}\}) \quad = \quad \neg\exists(\ \textcircled{x:i} \xrightarrow{k} \textcircled{y:i} \mid \texttt{atom(x, y) and int(i)}\ )$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\{\mathrm{Pre}(\texttt{init}, e)\}\ \texttt{init}\ \{e\}}[\text{ruleapp}]}{\{e\}\ \texttt{init}\ \{e\}}[\text{cons}]}{\{e\}\ \texttt{init!}\ \{e \wedge \neg\mathrm{App}(\{\texttt{init}\})\}}[!]}{\{c\}\ \texttt{init!}\ \{d\}}[\text{cons}] \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{\{\mathrm{Pre}(\texttt{inc}, d)\}\ \texttt{inc}\ \{d\}}[\text{ruleapp}]}{\{d\}\ \texttt{inc}\ \{d\}}[\text{cons}]}{\{d\}\ \texttt{inc!}\ \{d \wedge \neg\mathrm{App}(\{\texttt{inc}\})\}}[!]}{\vdash_{\text{par}} \{c\}\ \texttt{init!};\ \texttt{inc!}\ \{d \wedge \neg\mathrm{App}(\{\texttt{inc}\})\}}[\text{comp}]$$

$$\begin{aligned}
c &= \forall(\ \textcircled{\texttt{a}}_1, \exists(\ \textcircled{\texttt{a}}_1 \mid \texttt{atom(a)}\ ))\\
d &= \forall(\ \textcircled{\texttt{a}}_1, \exists(\ \textcircled{\texttt{a}}_1 \mid \texttt{a} = \texttt{b:c and atom(b) and c} >= 0))\\
e &= \forall(\ \textcircled{\texttt{a}}_1, \exists(\ \textcircled{\texttt{a}}_1 \mid \texttt{atom(a)}\ )\\
&\qquad \vee \exists(\ \textcircled{\texttt{a}}_1 \mid \texttt{a} = \texttt{b:c and atom(b) and c} >= 0))\\
\neg\mathrm{App}(\{\texttt{init}\}) &= \neg\exists(\ \textcircled{\texttt{x}}\ \mid \texttt{atom(x)}\ )\\
\neg\mathrm{App}(\{\texttt{inc}\}) &= \neg\exists(\ \textcircled{\texttt{x:i}} \xrightarrow{\texttt{k}} \textcircled{\texttt{y:i}}\ \mid \texttt{atom(x, y) and int(i)}\ )
\end{aligned}$$

Next on the agenda

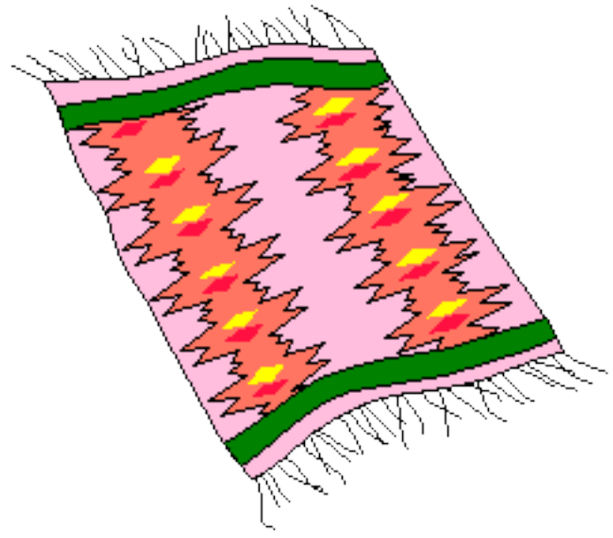
(1) a programming language for graphs

(2) an assertion language for graphs

(3) Hoare-style reasoning about graph transformation

(4) program proofs

The full picture



- many technical details hidden "under the carpet"

  - impossible to cover everything in the assigned time

- the full picture is quite interesting (I think!)

  - references will be added to the course webpage
  - but these are of course optional readings
  - the exercises on Wednesday will make clear the level of understanding I aimed for

Summary

- motivated the study of graph-manipulating programs and discussed some applications
- introduced the notion of graph transformation: program states as graphs; steps as rules
- considered a programming language for modelling problems as graph transformations
- presented an overview of an assertion language and Hoare logic for proving properties about graph structure and relations between labels

Ongoing work

- reasoning about arbitrary-length path properties

- graph-based semantics for concurrency models

Thank you! Questions?

Next lecture:

- data flow analysis (with Sebastian Nanz)