Gentle Introduction to LMNtal: Language Design and Implementation

— GT from a PL perspective

Kazunori Ueda

Waseda University, Tokyo, Japan

ICGT 2024, Twente, July 10, 2024

LMNtal (pronounce: “elemental”)

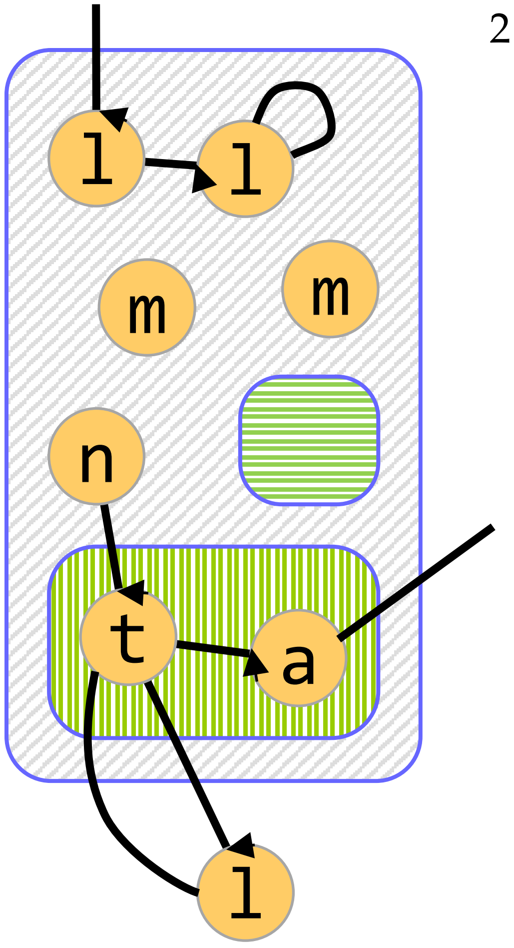
$\mathcal{L}$ = “logical” links

$\mathcal{M}$ = multisets/membranes

$\mathcal{N}$ = nested nodes

ta = transformation

ℓ = language



“node-labelled, open, hierarchical, and undirected port multigraph (or multihypergraph)”

More info about LMNtal in the LMNtal webpage, WMC 2004 (LNCS 3365), RTA 2008 (LNCS 5117), TCS **410** (2009), ICGT 2019/2023, GitHub, etc.

LMNtal: a unifying language

- **Project LMNtal** (pronounce “elemental”)
  - A *“unifying”* computational model + language + system based on (a class of) *graph rewriting*
  - Now 4th-generation implementation
  - >100,000 LOC involving many people over the years
  - Features *verification (model checking)* since 2007
  - Provides LaViT, an IDE with *visualizers*

Ready to use; very low entry barrier

http://www.ueda.info.waseda.ac.jp/lmntal/

- open-source from GitHub

K. Ueda, *Theoretical Computer Science* **410**, 2009
K. Ueda et al., *Proc. ICTAC 2009*, LNCS 5684

What do we mean by “unifying” ?

Unify various programming concepts, e.g.:

- data and functions
- processes and messages

(they just react!)

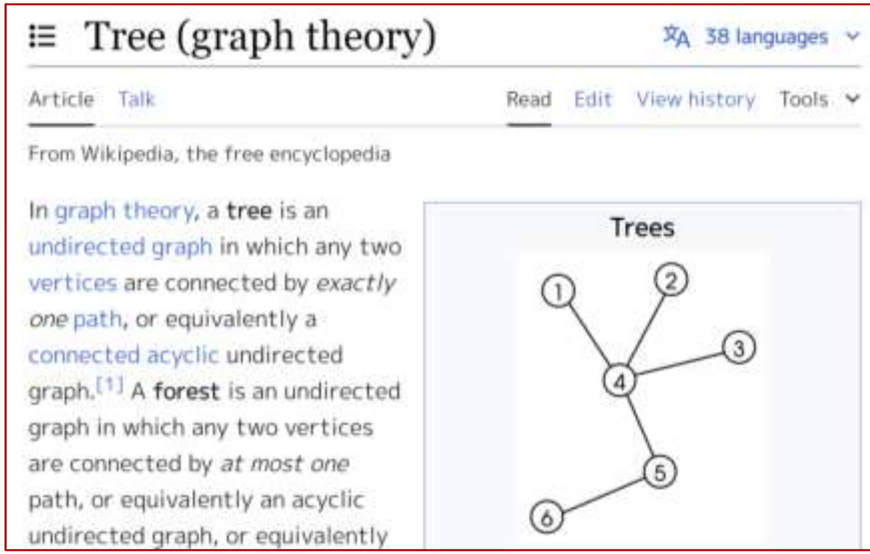
- data structures and process structures
  - Most *declarative* languages are awkward in handling non-tree (= non-algebraic) data structures
  - Pointers in *imperative* languages are error-prone
- synchronous and asynchronous communication
- programming and modeling
- computation and verification

What do we mean by a “PL perspective”?

- The primary concern is to have
  - *inductively defined* syntax and
  - *syntax-directed* semantics (= structural operational semantics)

  for graphs and graph transformation
  - *Analogy*:
    proof theory (vs. model theory) in mathematical logic
- Other PL concerns and interests include:
  - composition
  - abstraction
  - encoding (of other calculi)

GT from a PL perspective, diagrammatically



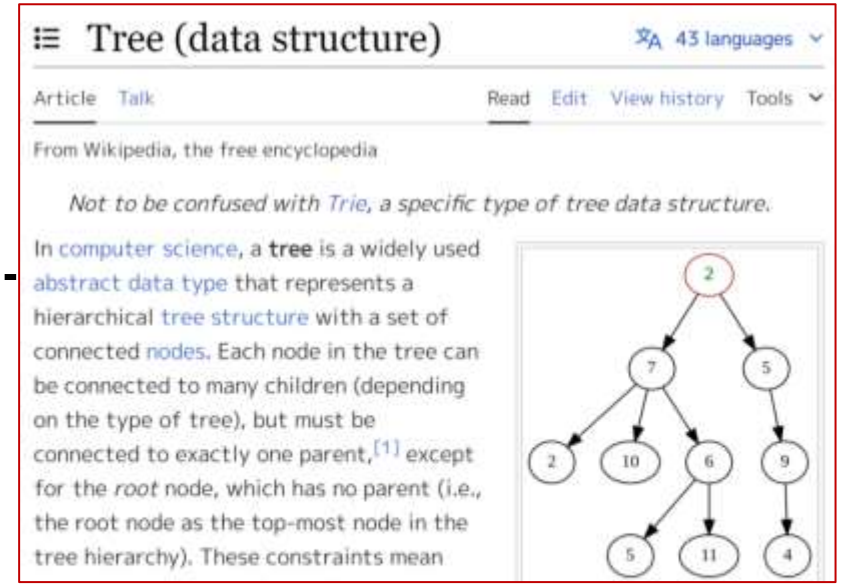
Tree (graph theory) 38 languages

Article Talk Read Edit View history Tools

From Wikipedia, the free encyclopedia

In graph theory, a **tree** is an undirected graph in which any two vertices are connected by *exactly one* path, or equivalently a connected acyclic undirected graph.[1] A **forest** is an undirected graph in which any two vertices are connected by *at most one* path, or equivalently an acyclic undirected graph, or equivalently

Trees



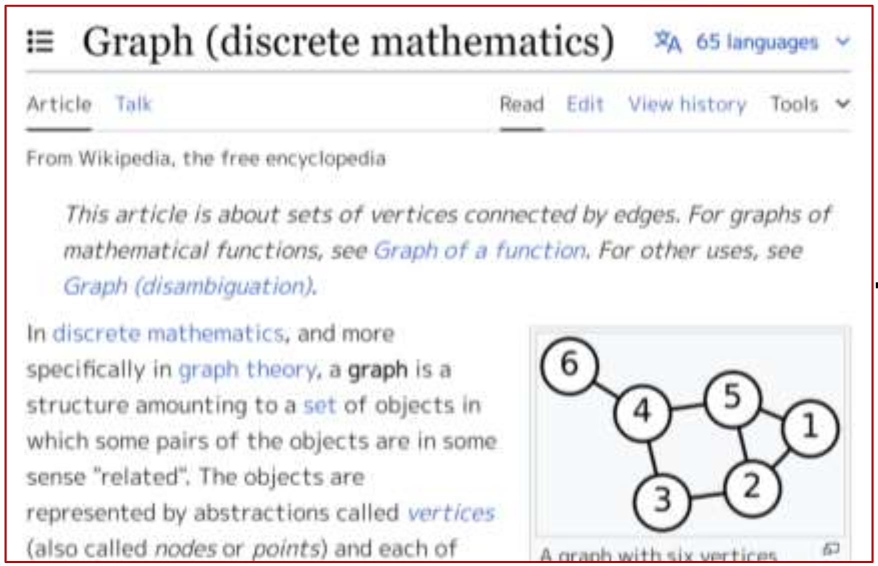
Tree (data structure) 43 languages

Article Talk Read Edit View history Tools

From Wikipedia, the free encyclopedia

Not to be confused with Trie, a specific type of tree data structure.

In computer science, a **tree** is a widely used abstract data type that represents a hierarchical tree structure with a set of connected nodes. Each node in the tree can be connected to many children (depending on the type of tree), but must be connected to exactly one parent,[1] except for the *root* node, which has no parent (i.e., the root node as the top-most node in the tree hierarchy). These constraints mean



Graph (discrete mathematics) 65 languages

Article Talk Read Edit View history Tools

From Wikipedia, the free encyclopedia

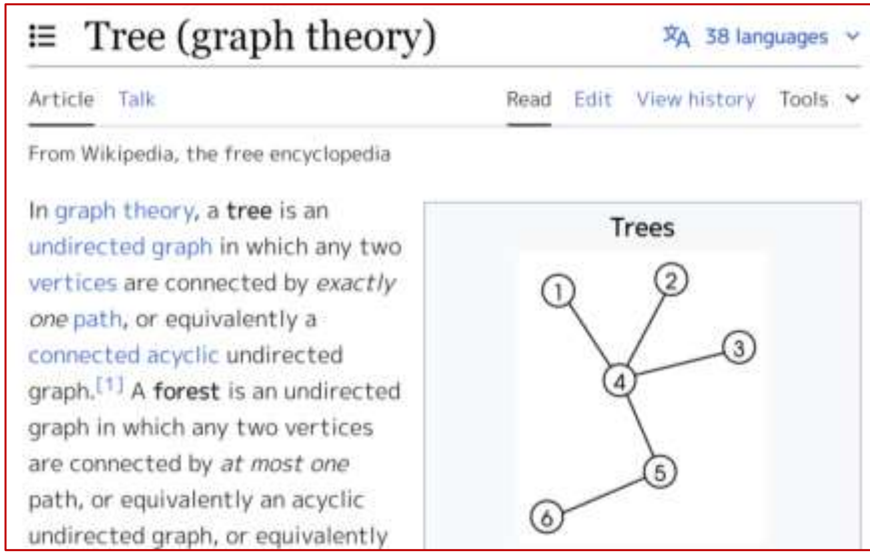
This article is about sets of vertices connected by edges. For graphs of mathematical functions, see Graph of a function. For other uses, see Graph (disambiguation).

In discrete mathematics, and more specifically in graph theory, a **graph** is a structure amounting to a set of objects in which some pairs of the objects are in some sense "related". The objects are represented by abstractions called *vertices* (also called *nodes* or *points*) and each of

A graph with six vertices

?

GT from a PL perspective, diagrammatically


Tree (graph theory)

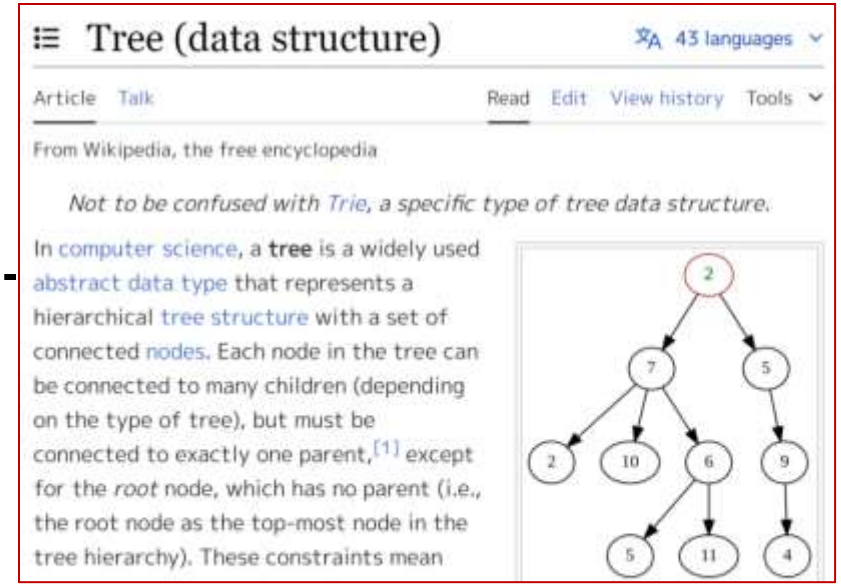
38 languages

Article Talk Read Edit View history Tools

From Wikipedia, the free encyclopedia

In graph theory, a **tree** is an undirected graph in which any two vertices are connected by *exactly one* path, or equivalently a connected acyclic undirected graph.[1] A **forest** is an undirected graph in which any two vertices are connected by *at most one* path, or equivalently an acyclic undirected graph, or equivalently

Trees


Tree (data structure)

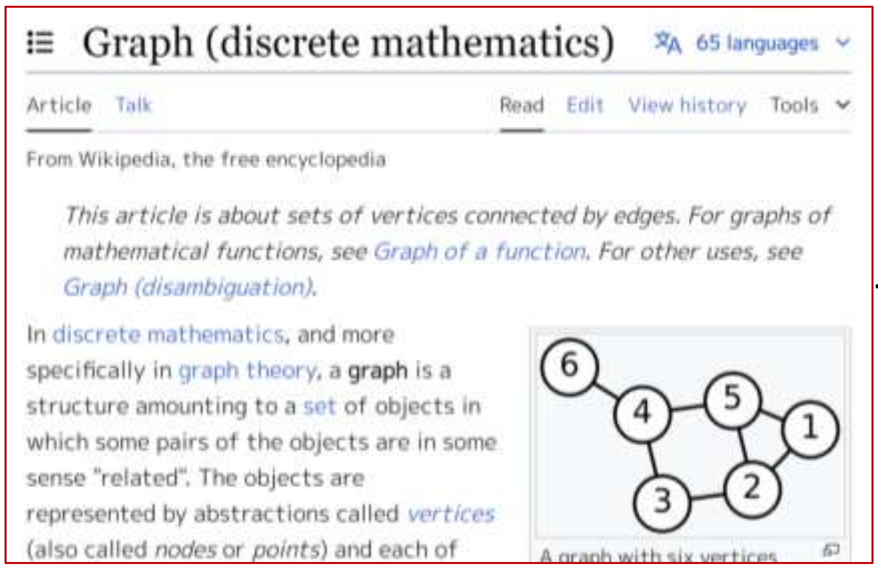
43 languages

Article Talk Read Edit View history Tools

From Wikipedia, the free encyclopedia

Not to be confused with Trie, a specific type of tree data structure.

In computer science, a **tree** is a widely used abstract data type that represents a hierarchical tree structure with a set of connected nodes. Each node in the tree can be connected to many children (depending on the type of tree), but must be connected to exactly one parent,[1] except for the *root* node, which has no parent (i.e., the root node as the top-most node in the tree hierarchy). These constraints mean


Graph (discrete mathematics)

65 languages

Article Talk Read Edit View history Tools

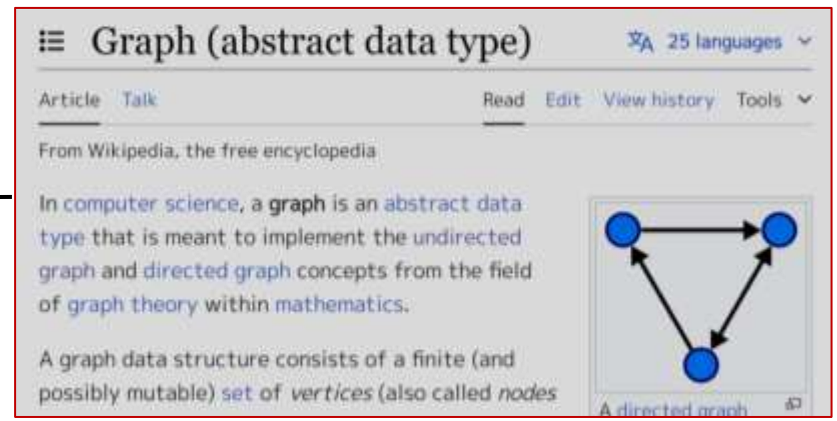
From Wikipedia, the free encyclopedia

This article is about sets of vertices connected by edges. For graphs of mathematical functions, see Graph of a function. For other uses, see Graph (disambiguation).

In discrete mathematics, and more specifically in graph theory, a **graph** is a structure amounting to a set of objects in which some pairs of the objects are in some sense "related". The objects are represented by abstractions called *vertices* (also called *nodes* or *points*) and each of

A graph with six vertices

$\cong$


Graph (abstract data type)

25 languages

Article Talk Read Edit View history Tools

From Wikipedia, the free encyclopedia

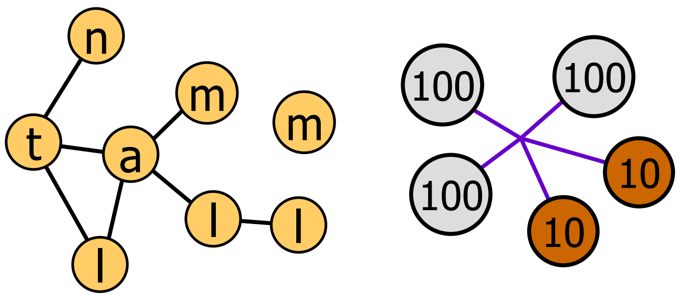
In computer science, a **graph** is an abstract data type that is meant to implement the undirected graph and directed graph concepts from the field of graph theory within mathematics.

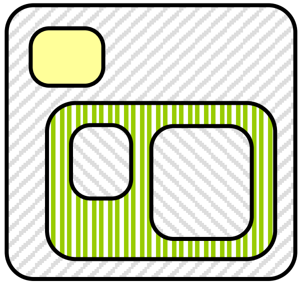
A graph data structure consists of a finite (and possibly mutable) set of *vertices* (also called *nodes*

A directed graph

Hierarchical (hyper)graphs: Motivations

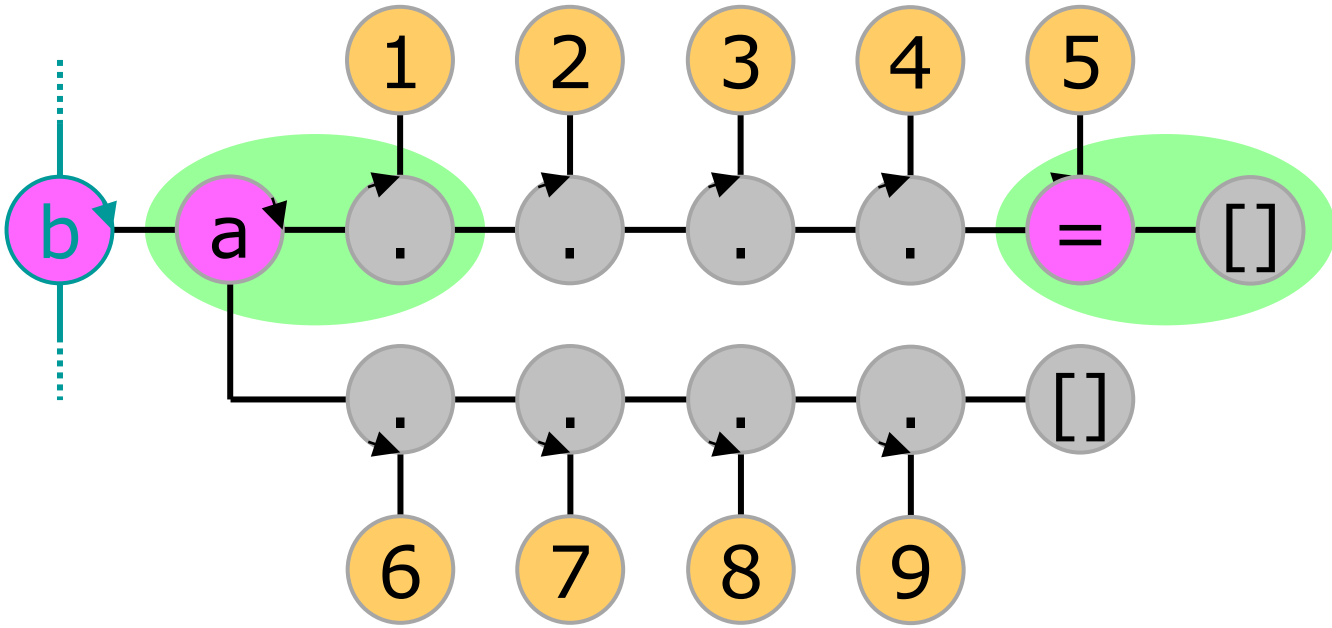
- Structures found in organization (of any kind) and human knowledge have one or both of the following:
  - **connectivity**
    - network, graphs, human relationships, ...
  - **hierarchy**
    - companies, addresses, domain names, ...

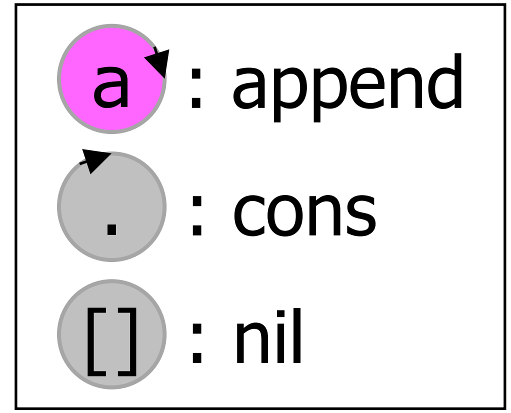


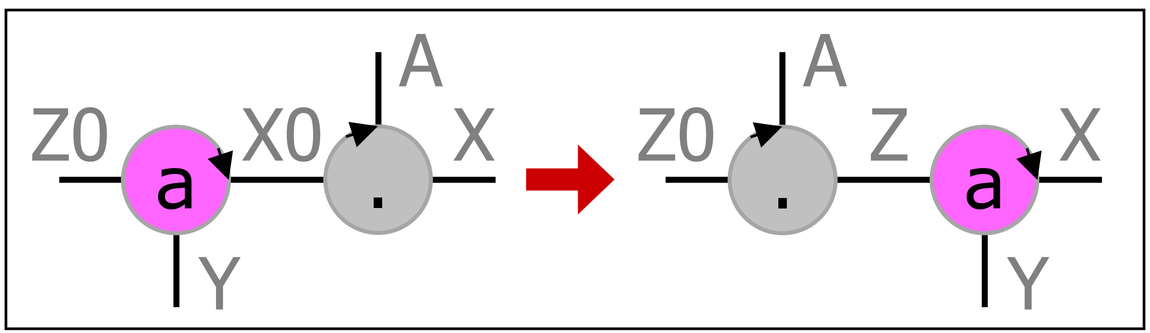


RQ: Can we have a concise programming language that allows us to represent and manipulate them simultaneously and in a direct, safe manner?

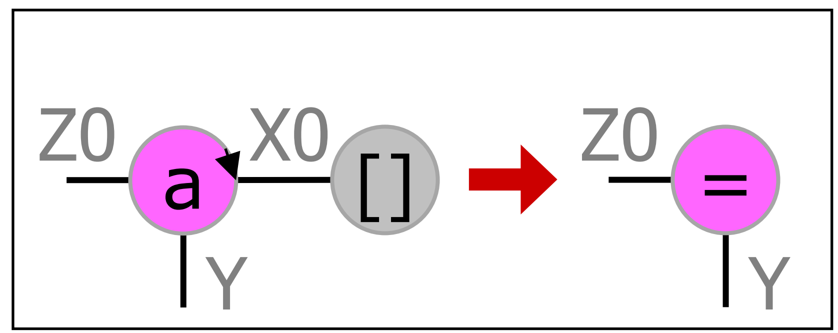
Example: List concatenation (*a la* Interaction Nets)





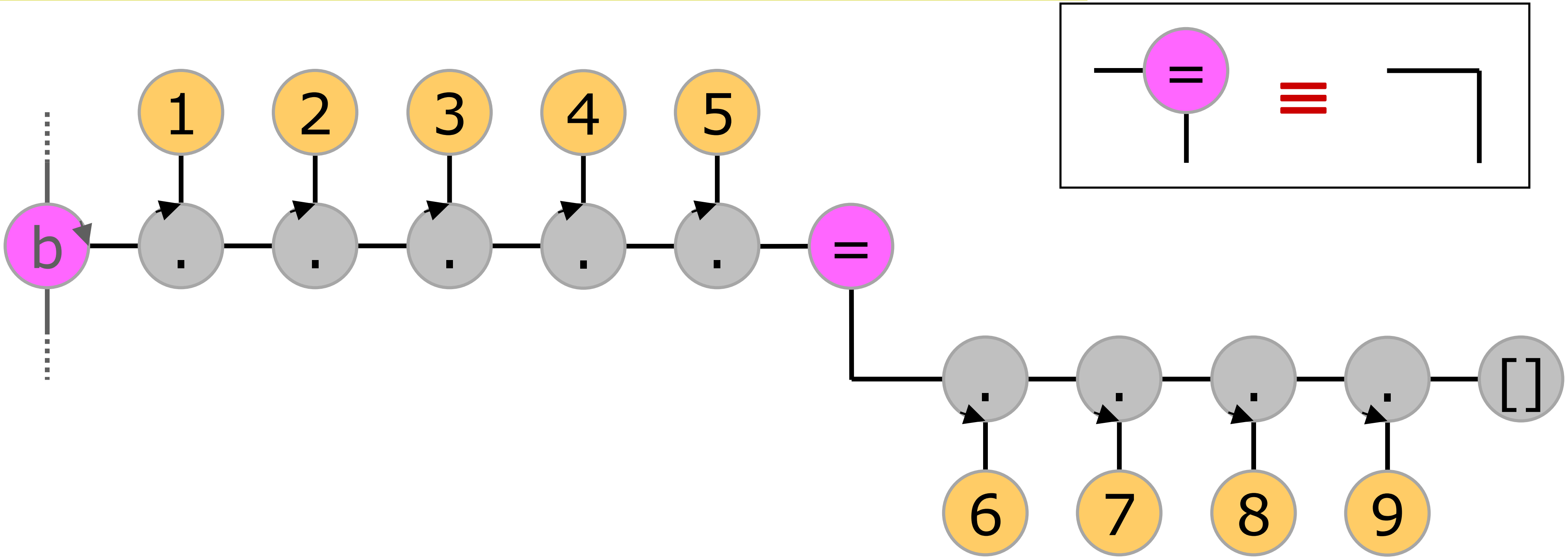


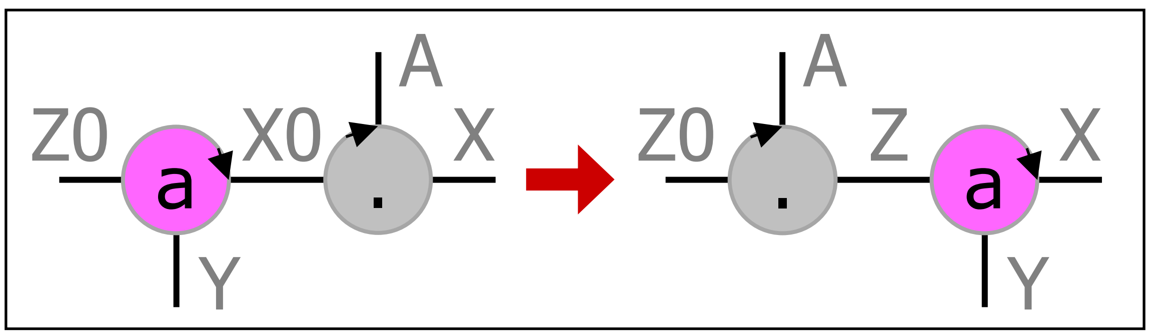
a(X0,Y,Z0), '.'(A,X,X0) :- '.'(A,Z,Z0), a(X,Y,Z)

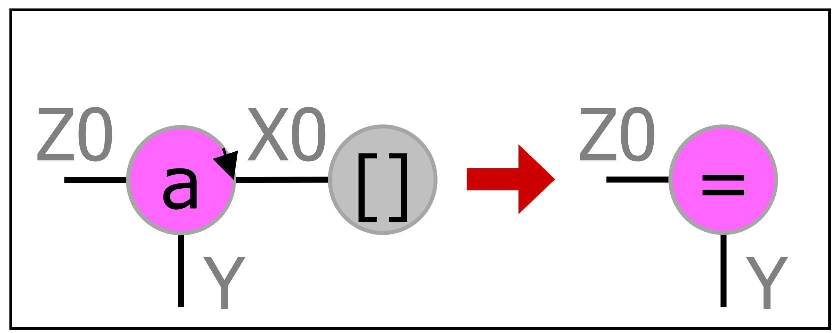


a(X0,Y,Z0), '[]'(X0) :- Y=Z0

Example: List concatenation (*a la* Interaction Nets)







a(X0,Y,Z0), '.'(A,X,X0) :- '.'(A,Z,Z0), a(X,Y,Z)

a(X0,Y,Z0), '[]'(X0) :- Y=Z0

append in LMNtal

```
append(X0,Y,Z0), '[]'(X0) :- Y=Z0
append(X0,Y,Z0), '.'(A,X,X0) :- '.'(A,Z,Z0), append(X,Y,Z)
```

- Constructors ('.' and '[]') are in relational form, but LMNtal provides a *term (or functional) notation*:

Expand (E9) Embed

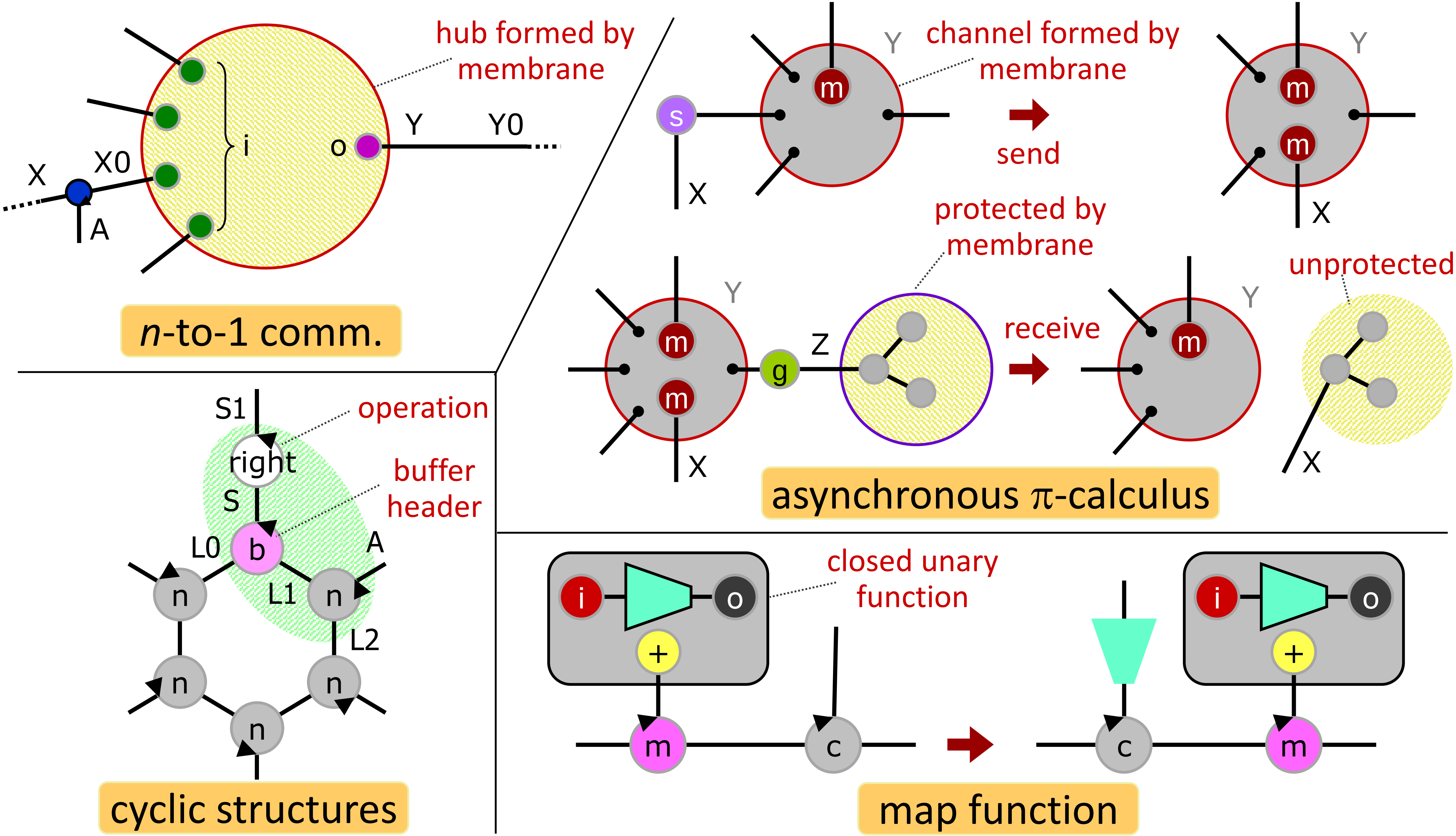
'[]'(X0) ≡ X0=Y, '[]'(Y) ≡ X0='[]'

'.'(A,X,X0) ≡ X0=Y, '.'(A,X,Y) ≡ X0='.'(A,X)

```
append(X0,Y,Z0), X0='.'(A,X) :- Z0='.'(A,Z), append(X,Y,Z)
```

```
Z0 = append('.'(A,X),Y)  :-  Z0 = '.'(A,append(X,Y))
```

Diagrammatic representation of computation



LMNtal in a nutshell

- A rule-based concurrent **language** for expressing and rewriting connectivity and hierarchy
- Unifying **model** of X-calculi ($X = \lambda, \pi$, ambient, etc.) and multiset rewriting
- Computation is manipulation of diagrams
  - Links express 1-to-1 connectivity
  - Membranes express hierarchy and locality
  - Allows programming with sets and graphs and programming by self-organization
  - Well-defined notion of atomic actions

Historical Background

- Logic Programming (early 1970's)
  - Procedural interpretation of logical formulae $(h \Leftarrow B)$
- Concurrent Logic Programming (early 1980's)
  - Process interpretation of logical formulae
  - *Channel mobility* using logical variables
- Constraint-Based Concurrency (late 1980's)
  - Generalization of data domains (FD, multisets, . . .)
- CHR (Constraint Handling Rules) (early 1990's)
  - Allows *multisets of goals* in rule heads
  - An expressive *multiset rewriting* language
  - Many applications (esp. constraint solvers)

Models and languages with multisets and *symmetric join*

- (Colored) Petri Nets
- Production Systems and RETE match
- Graph transformation formalisms
- CCS, CSP
- Concurrent logic/constraint programming
- Linda
- Linear Logic languages
- Interaction Nets
- Chemical Abstract Machine, reflexive CHAM, Join Calculus
- Gamma mode
- Maude
- Constraint Handling Rules (CHR)
- Mobile ambients
- P-system, membrane computing
- Amorphous computing
- Bigraphical Reactive Systems

Models and languages with *membranes + hierarchies*

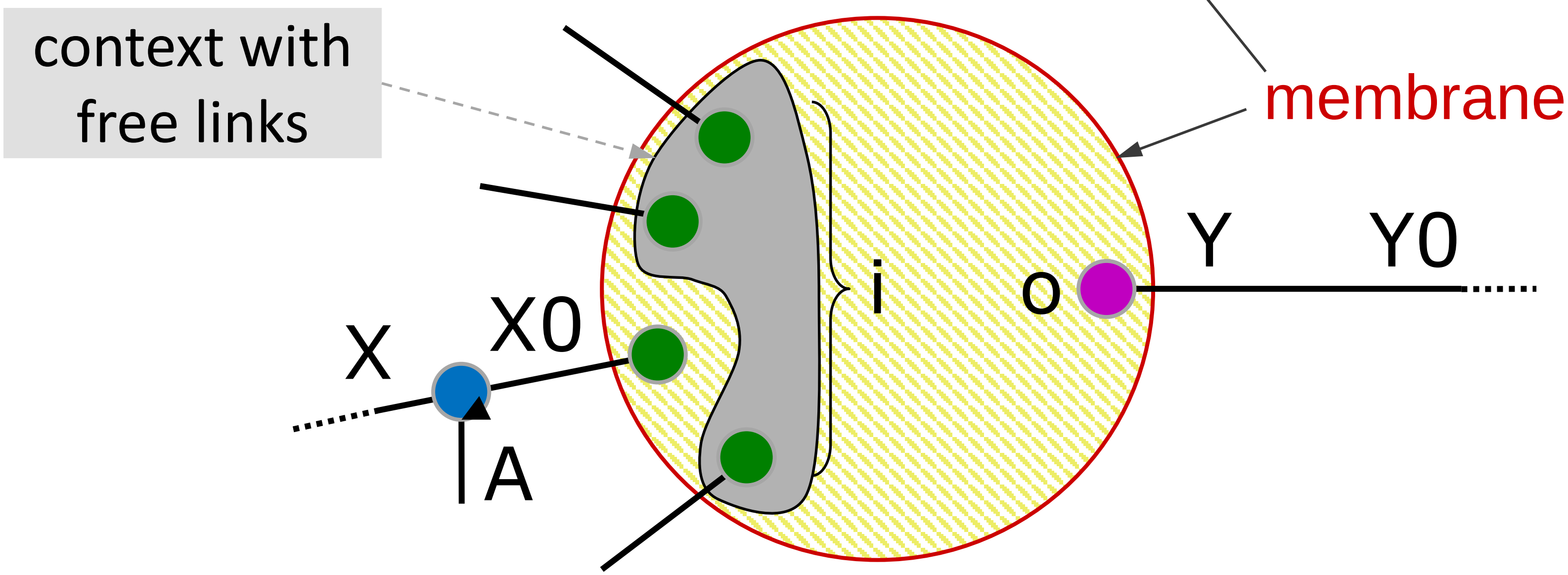
- (Colored) Petri Nets
- Production Systems and RETE match
- Graph transformation formalisms *
- CCS, CSP
- Concurrent logic/constraint programming
- Linda *
- Linear Logic languages
- Interaction Nets
- Chemical Abstract Machine, reflexive CHAM, Join Calculus
- Gamma model
- Maude
- Constraint Handling Rules
- Mobile ambients
- P-system, membrane computing
- Amorphous computing
- Bigraphical Reactive Systems

* : some versions feature hierarchies

- Statecharts
- Seal calculus
- Kell calculus
- Brane calculi
- κ calculus

Example: N-to-1 stream/channel communication

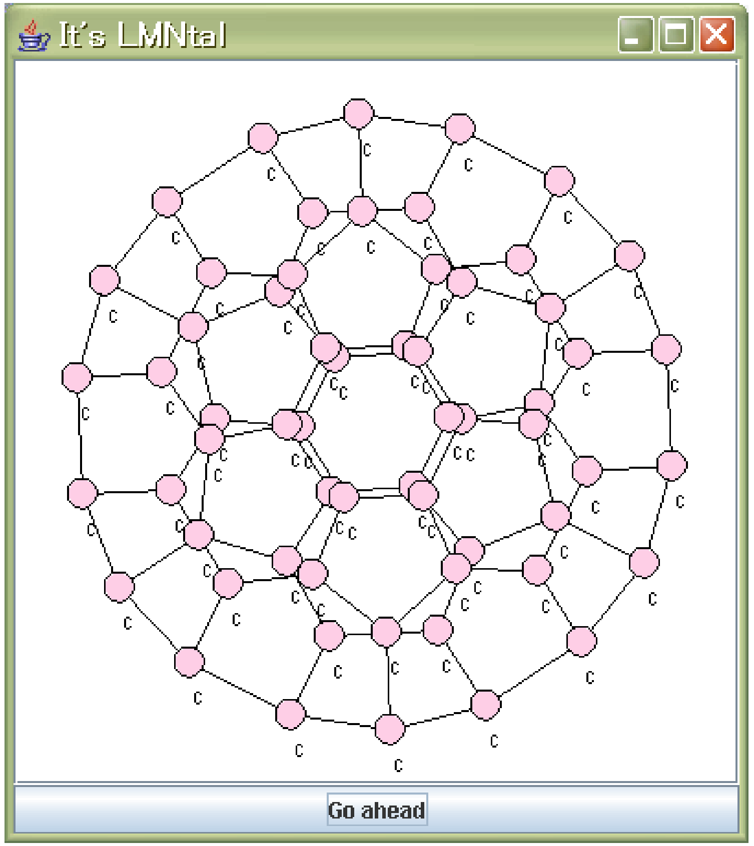
```
c(A,X,X0), { i(X0), o(Y0), $p } :-
              { i(X), o(Y), $p }, c(A,Y,Y0)
```



- The number of free links of { } remain unchanged

Demo: Fullerene (C_{60})

```
 /* icosahedron */
dome(L0,L1,L2,L3,L4,L5,L6,L7,L8,L9) :-
   p(T0,T1,T2,T3,T4),  p(L0,L1,H0,T0,H4),
   p(L2,L3,H1,T1,H0),  p(L4,L5,H2,T2,H1),
   p(L6,L7,H3,T3,H2), p(L8,L9,H4,T4,H3).

dome(E0,E1,E2,E3,E4,E5,E6,E7,E8,E9),
dome(E0,E9,E8,E7,E6,E5,E4,E3,E2,E1).

/* icosahedron -> fullerene */
p(L0,L1,L2,L3,L4) :-
  c(L0,X0,X4), c(L1,X1,X0), c(L2,X2,X1), c(L3,X3,X2), c(L4,X4,X3).
```



Elements of LMNtal (1+2)

1+2. Labelled nodes (called *atoms*) with ordered links

- Origin: atomic formula $p(t_1, \ldots, t_n)$ in first-order logic
- An atom has its own arity (degree), and its links are totally ordered (cf. graphs in graph theory)
  - *a.k.a.* “(node of) port graph”, “hyperedge”
- Links are linear, *zero-assignment* logical variables
  - linear = occurring twice (1-to-1 communication)
  - zero-assignment = not instantiated to terms
  - logical (*a.k.a.* immutable) = link identity changes after message sending (cf. π-calculus)
  - not directed (like chemical bonds)

Elements of LMNtal (1+2) (cont'd)

- Links are used for :
  - (a) representing (private) *communication channels*
  - (b) forming *complex data structures* (i.e., graphs)
  - (c) finding *partners in rewriting*
    - ✓ O(1) if linked
    - ✓ can be O(*n*) if not linked
  - (d) representing *hyperlinks* (using *membranes* (p.21))
- HyperLMNtal (2011) features *hyperlinks* as an independent construct

Elements of LMNtal (3)

3. Membranes (to represent *first-class multisets*)

- Not many languages feature multisets (or forests) as *first-class* citizens
- Used for :
  - representing *records* (*a.k.a.* feature structures, KVS)
  - representing *graph nodes with variable degrees*
  - localization and management of computation
    - ✓ cf. ambients, join calculus, Unix processes
  - creating and managing *fresh local names*

Elements of LMNtal (4)

4. **Rewrite rules**

- Can be placed in a membrane to realize
  - local reaction
  - mobility of autonomous processes
- Key design issue: proper handling of *free links* (*a.k.a. open/half/dangling edges, loose ends,* and *links with boundary/exterior vertices*)
  - Recall: LMNtal handles open graphs (*a.k.a. semigraphs*)
  - design of context handling is the key

Syntax: preliminaries

- Two presupposed syntactic categories:
  - X : link names (linear (1-to-1) & local)
    - In concrete syntax, start with capital letters
  - p : atom names (nonlinear & global)
    - Works as node labels
    - In concrete syntax, use identifiers different from links (e.g., `cons`, `314`, `'+'`, `"string"`)
- Atom names are uninterpreted, except for '=' (called a connector)

Syntax of LMNtal processes

- $P \ ::= \ \mathbf{0}$ (null)
  $\quad | \quad p(X_1, \ldots, X_m) \ \ (m \geq 0)$ (atom)
  $\quad | \quad P, P$ (molecule)
  $\quad | \quad \{P\}$ (cell)
  $\quad | \quad T \ {:-} \ T$ (rule)

Not in Flat LMNtal

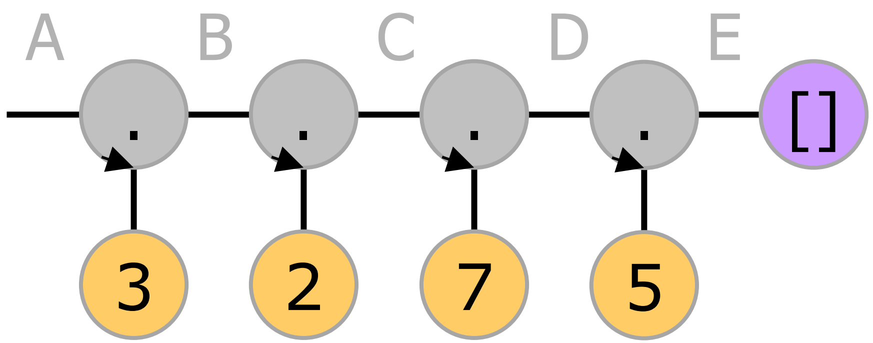
- **Link condition:** Each link name in P occurs at most twice and each link name in a rule occurs exactly twice.
  - Free link of P = link occurring only once
  - P is closed = has no free links

Syntax of LMNtal process templates

Not in Flat LMNtal

- $T ::= \mathbf{0}$ (null)
  - $\mid\ p(X_1,\ldots,X_m)\ (m \geq 0)$ (atom)
  - $\mid\ T, T$ (molecule)
  - $\mid\ \{T\}$ (cell)
  - $\mid\ T \text{ :- } T$ (rule)
  - $\mid\ @p$ (rule context)
  - $\mid\ \$p[X_1,\ldots,X_m \mid A]\ (m \geq 0)$ (process context)
  - $\mid\ p(*X_1,\ldots,*X_m)\ (m \geq 0)$ (aggregate)
- (residual args) $A ::= [\,]$ (empty)
  - $\mid\ *X$ (bundle)

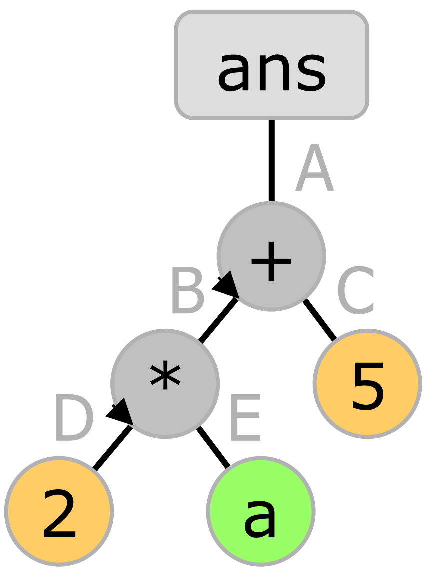
Lists, trees, bags (*cells*)



'.'(3,B,A), '.'(2,C,B),
'.'(7,D,C), '.'(5,E,D), '[]'(E)

- or -

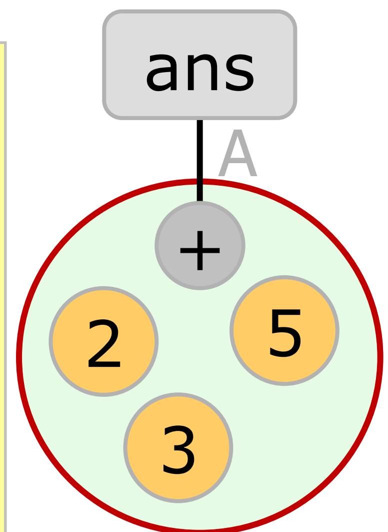
A = '.'(3, '.'(2, '.'(7, '.'(5,'[]'))))
A = [3 | [2 | [7 | [5 | []]]]]
A = [3, 2, 7, 5]



ans(A),
'+'(B,C,A), '*'(D,E,B),
2(D), a(E), 5(C)

- or -

ans('+'('*'(2, a), 5))
ans = 2*a+5



ans(A),
{+A, 2, 3, 5}

- or -

ans({2, 3, 5})
ans={2, 3, 5}

Term notation

remove the final arg
and fold the rest

'.'(A1,X1,X0), '.'(A2,X2,X1), '.'(A3,X3,X2), '[]'(X3)

≡ '.'(A1, '.'(A2, '.'(A3, '[]')), X0)

expand using =

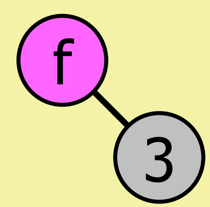
≡ X0 = Y, '.'(A1, '.'(A2, '.'(A3, '[]')),Y)

fold again

≡ X0 = '.'(A1, '.'(A2, '.'(A3, '[]')))

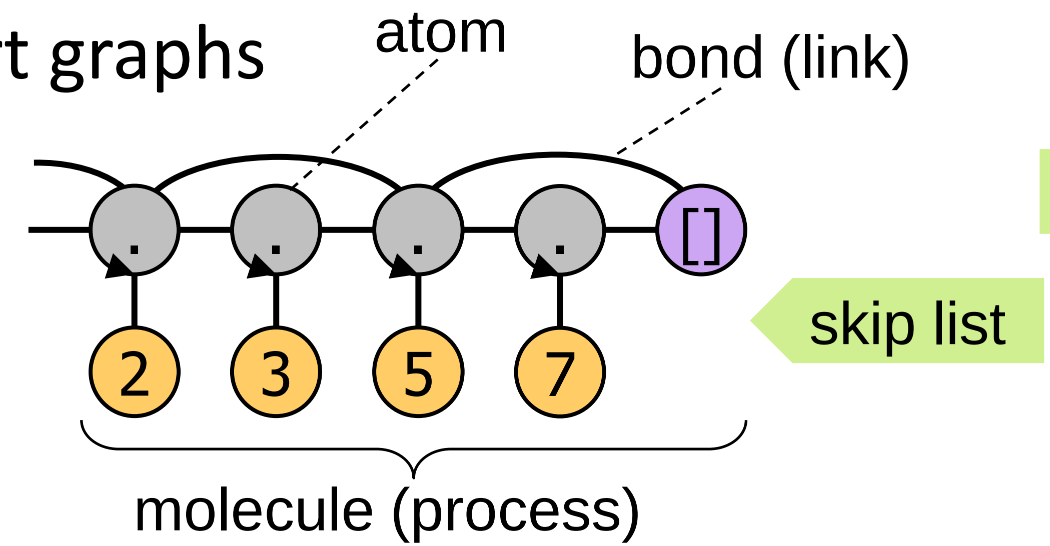
list notation

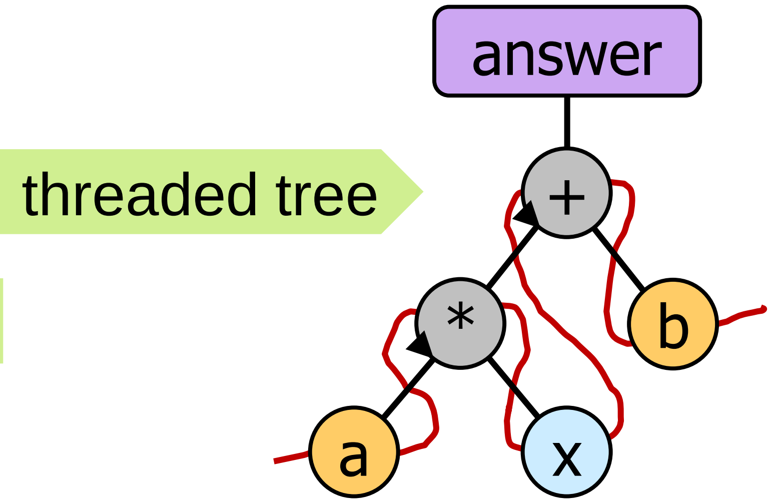
≡ X0 = [A1, A2, A3]

f(X),3(X) ≡ f(3) ≡ 3(f) ≡ f = 3 ≡ 3 = f ≡ 

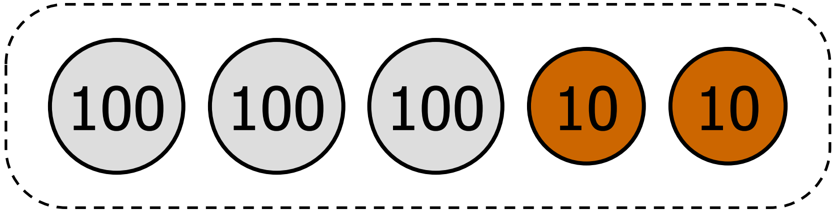
(Port) graphs and multisets

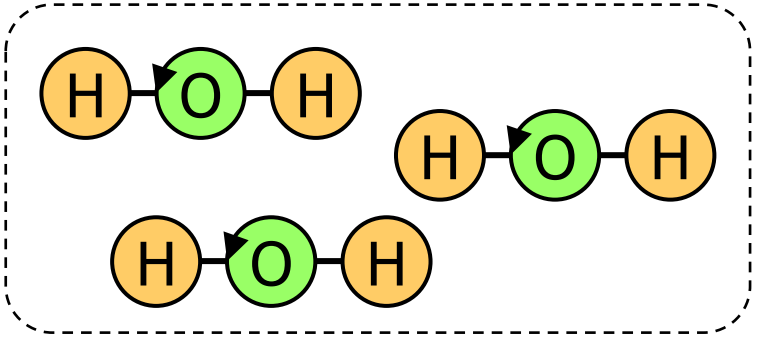
- Port graphs





- Multisets

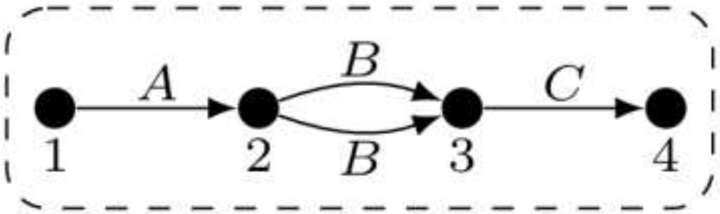




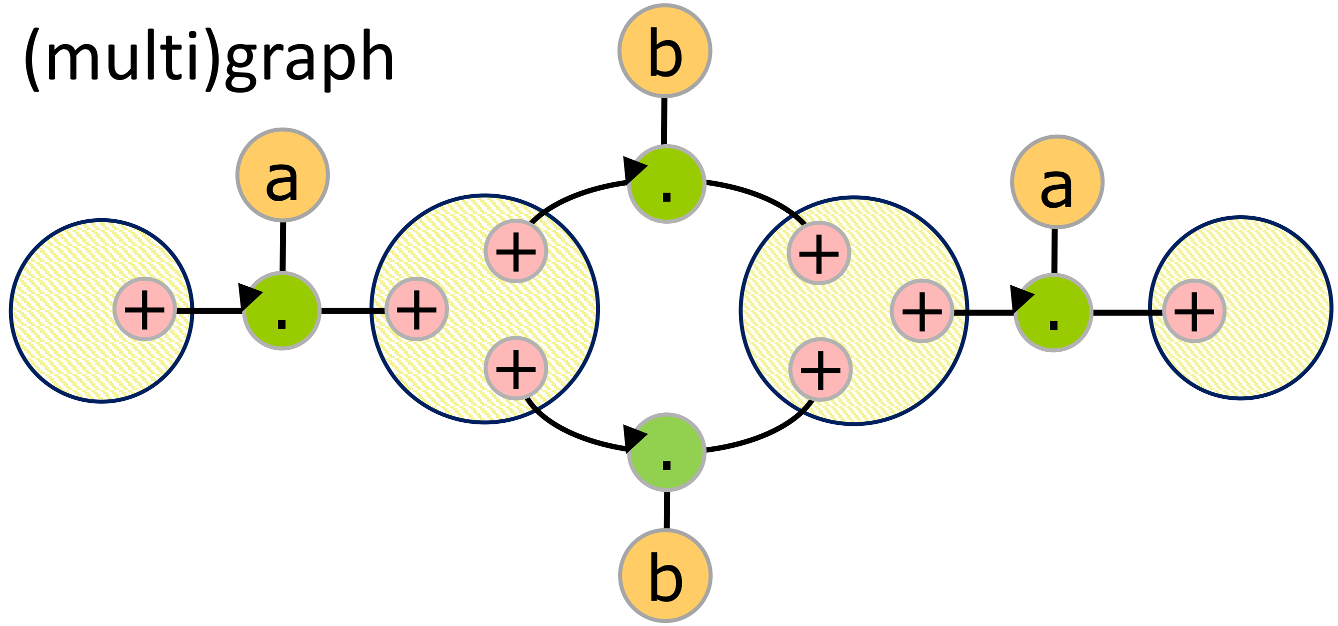
... are graphs with “less” edges (another important direction!)

How about “standard” graphs?

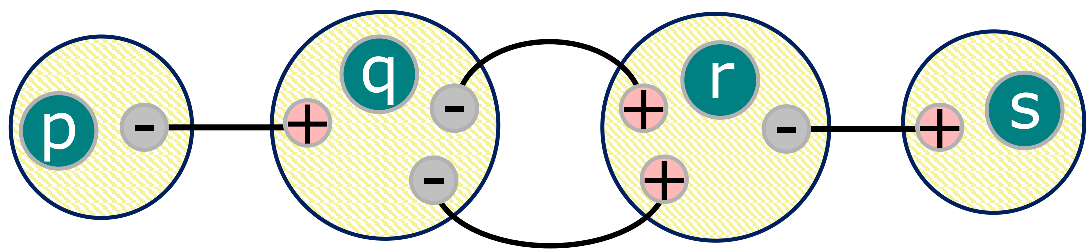
- Edge-labeled directed (multi)graph



König et al., LNCS 10800



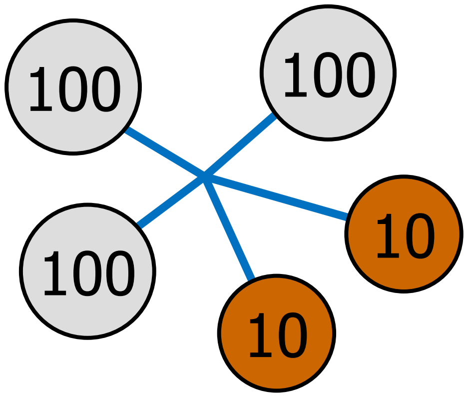
- Node-labeled directed (multi)graph
  (with several other
  alternatives depending
  on applications)



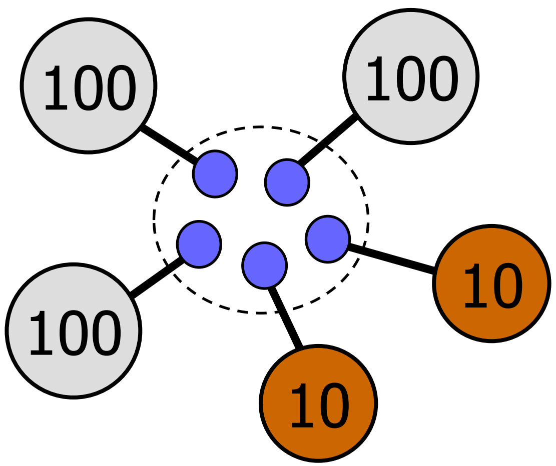
- Attributed graphs in a similar manner

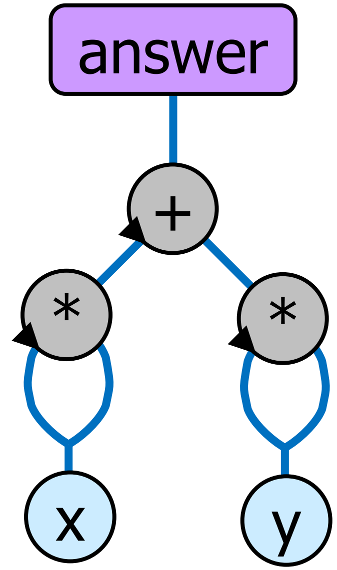
Hypergraph extension (HyperLMNtal)

- Hyperlinks (blue or curved) handled separately from ordinary links

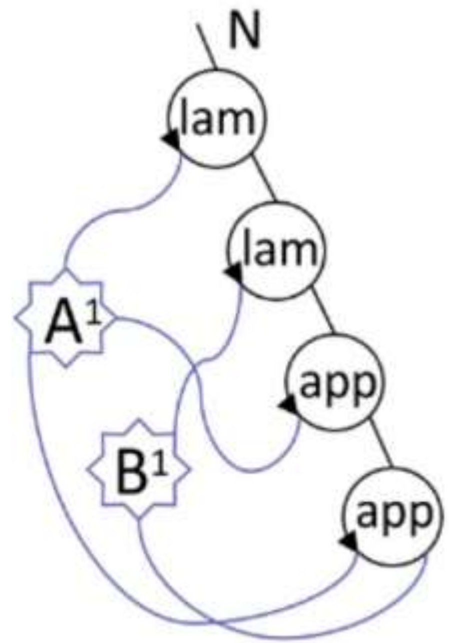


multiset





term graph



$\lambda f.\lambda x.f(f x)$

A hyperlink can be regarded as a collection of *unary atoms with a private name (and an optional attribute)*

How to associate “variable names” and data?

- Scalars: `x=20, y=8` *- or -* `x(20), y(8)`
- Records: `x=date(2024,july,10)`
- Lists: `x=[2,3,5,7]`
- Multisets: `x={n(2),n(2),n(3)}` (cf. `x={2,2,3}`)

- A name and its value are connected by a link, which means that each value has *one* free link (i.e., is *unary*)
  - Each value can be regarded as linked to its *owner* (cf. Rust)

Representing algorithms

- Euclid's algorithm

```
m=20, n=8.
m=$x, n=$y :- $x>$y | m=$x-$y, n=$y.
m=$x, n=$y :- $x<$y | m=$x, n=$y-$x.
```

  - The algorithm uses no procedure/function names (can be considered as reaction between data)

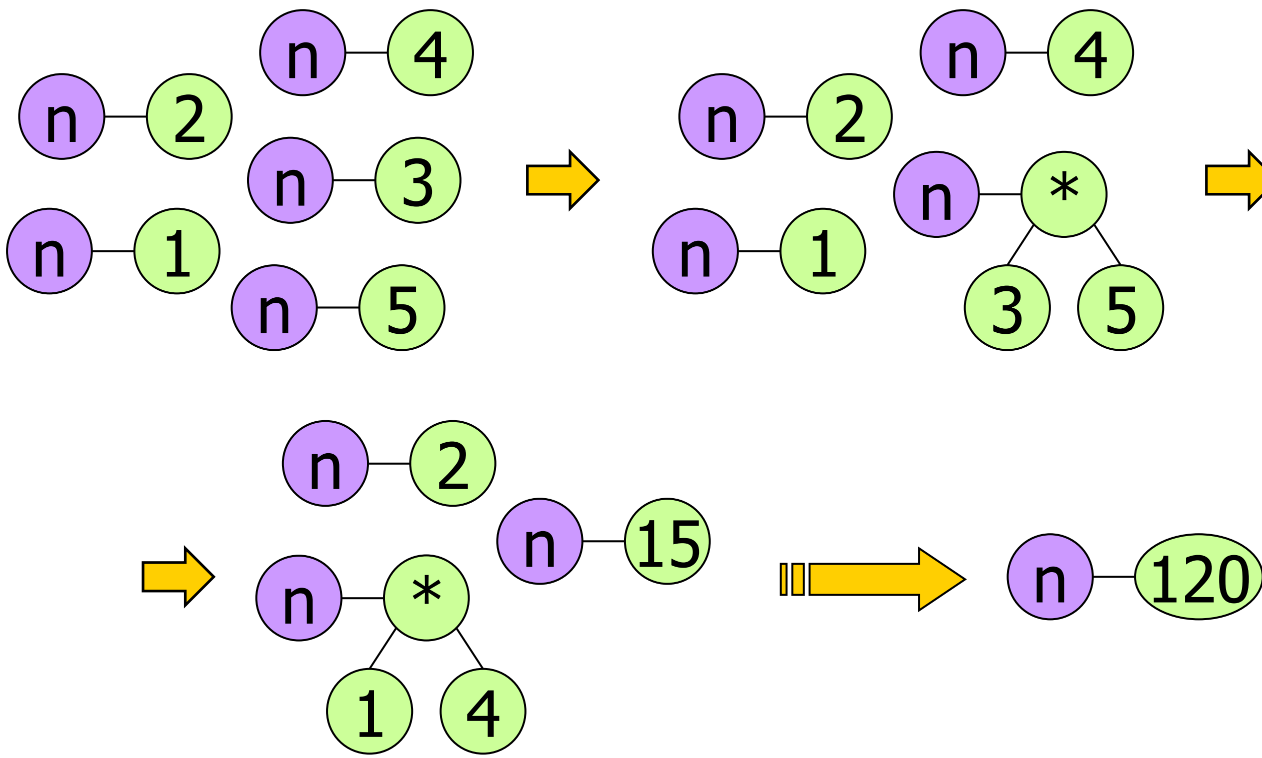
- LMNtal allows us to give identical names to two or more data (useful for *symmetric* algorithms)

```
n=20, n=8.
n=$x, n=$y :- $x>$y | n=$x-$y, n=$y.
```

Demo: Factorial

```
n(1), n(2), n(3), n(4), n(5).
n($a), n($b) :- n($a*$b).
n(A), $a[A], n(B), $b[B] :- n(E), '*'(C,D,E), $a[C], $b[D].
```

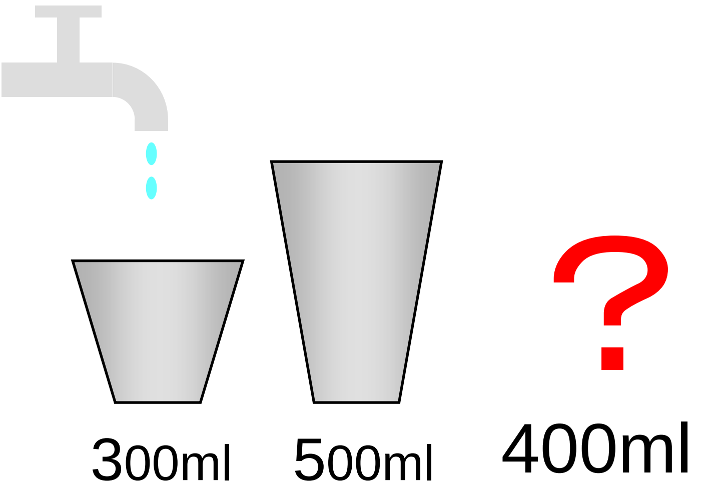
Numbers are unary atoms (= atom with a single link)

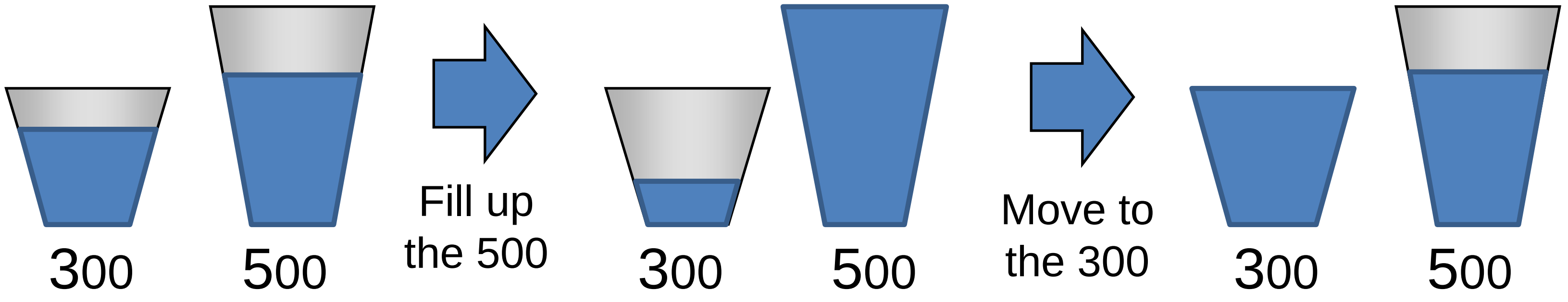


Arithmetics is handled by built-in rulesets

Demo: Water jug problem

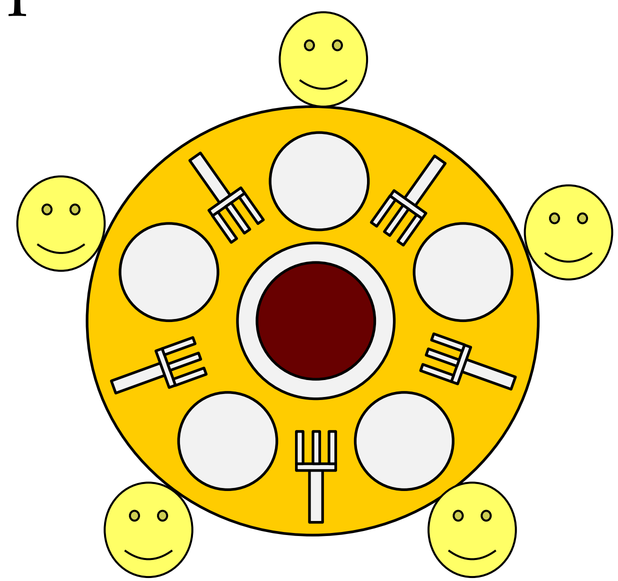
- Given a 300ml jug and a 500ml jug, get 400ml of water
- Allowed operations:
  - Empty a jug
  - Fill up a jug with tap water
  - Move a jug's water to the other until it's emptied
  - Move a jug's water until the other jug is filled up





Demo: Dining philosophers (due to E. W. Dijkstra)

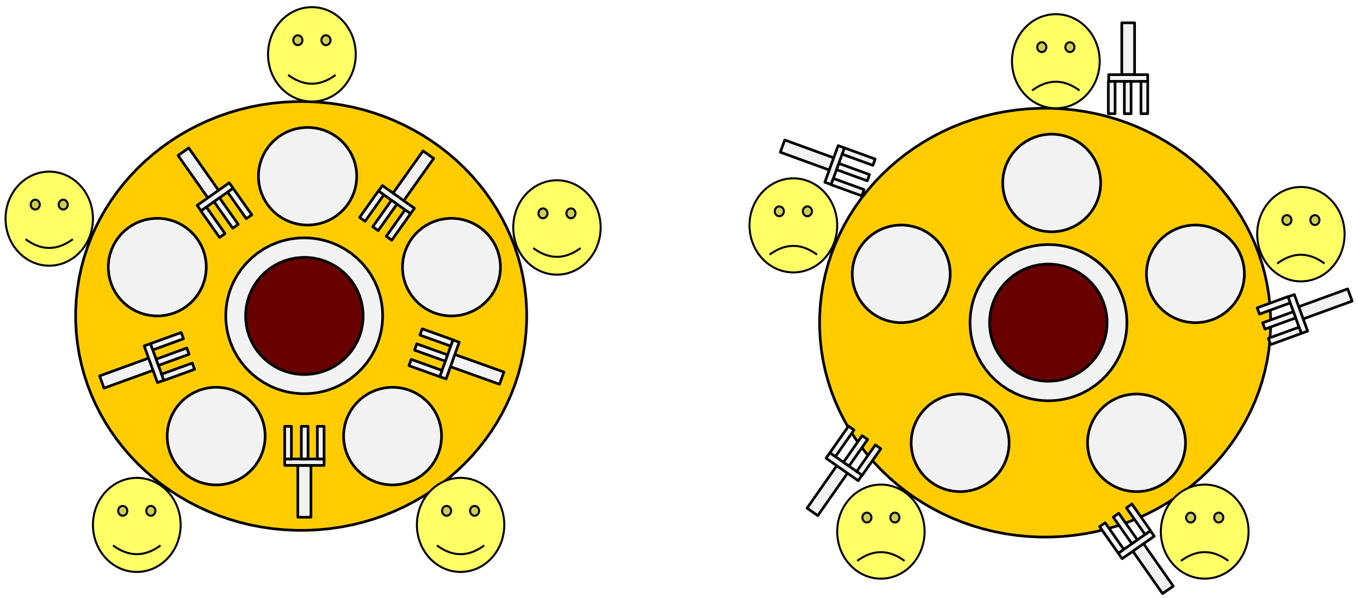
Five philosophers spend their lives thinking and eating. They share a common dining room where there is a circular table surrounded by five chairs, each belonging to one philosopher. In the center of the table there is a large bowl of spaghetti, and the table is laid with five forks. On feeling hungry, a philosopher enters the dining room, its in his own chair, and picks up the fork on the left of his plate. Unfortunately, the spaghetti is so tangled that he needs to pick up and use the fork on his right as well. When he has finished, he puts down both forks, and leaves the room.



— *C.A.R. Hoare (1978)*

Demo: Dining philosophers

- A flock of mediocre philosophers would cause deadlock . . .



. . . but a perverse philosopher avoids deadlock!

- See how symmetry is reduced in state-space search.

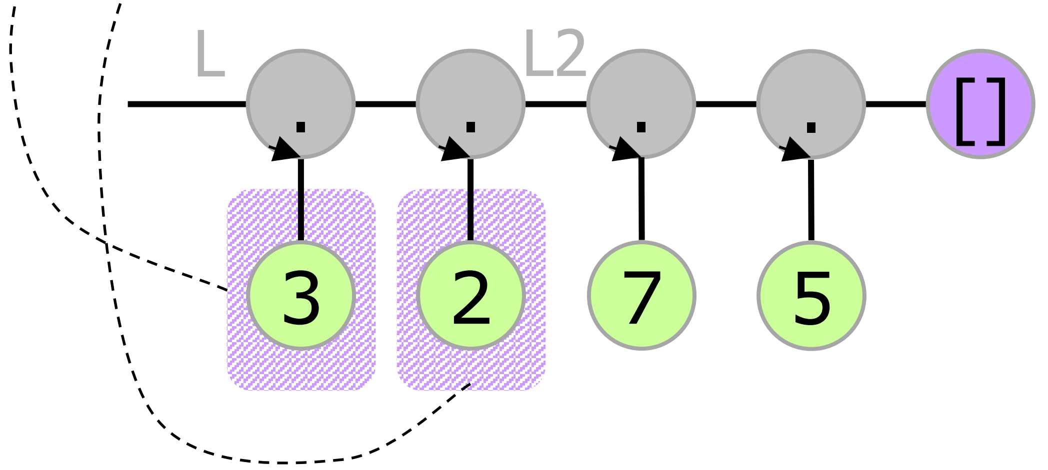
Demo: Bubblesort

typed process context

guard

```
L=[$x , $y|L2] :- $x > $y | L=[$y, $x|L2].
```

compare and swap if $x > $y



The left-hand side may match the middle of a list.

Structural congruence ($\equiv$)

multisets

(E1) $\mathbf{0}, P \equiv P$

(E2) $P, Q \equiv Q, P$

(E3) $P, (Q, R) \equiv (P, Q), R$

structural

(E4) $P \equiv P[Y/X]$ if X is a local link of P

(E5) $P \equiv P' \Rightarrow P, Q \equiv P', Q$

(E6) $P \equiv P' \Rightarrow \{P\} \equiv \{P'\}$

connectors

(E7) $X{=}X \equiv \mathbf{0}$

(E8) $X{=}Y \equiv Y{=}X$

(E9) $X{=}Y,\ P \equiv P[Y/X]$

if P is an atom and X is a free link of P

(E10) $\{X{=}Y,\ P\} \equiv \{P\},\ X{=}Y$

if exactly one of X and Y is a free link of P

Structural congruence, pictorially

connectors

(E7) $X{=}X \;\equiv\; \mathbf{0}$

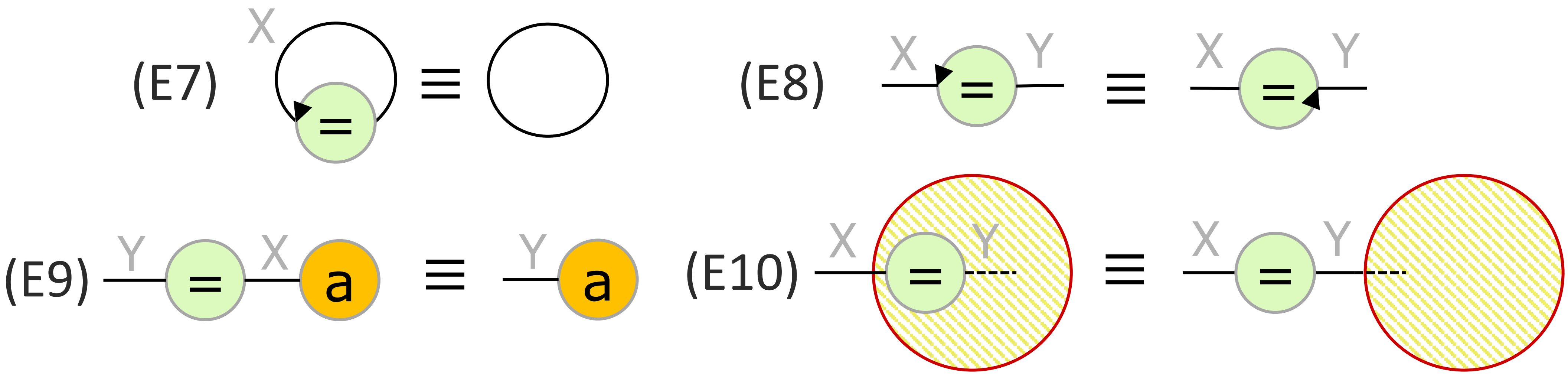
(E8) $X{=}Y \;\equiv\; Y{=}X$

(E9) $X{=}Y,\, P \;\equiv\; P[Y/X]$

if P is an atom and X is a free link of P

(E10) $\{X{=}Y,\; P\} \;\equiv\; \{P\},\, X{=}Y$

if exactly one of X and Y is a free link of P



Structural congruence, notes

multisets

(E1) $\mathbf{0}, P \equiv P$

(E2) $P, Q \equiv Q, P$

(E3) $P, (Q, R) \equiv (P, Q), R$

structural

(E4) $P \equiv P[Y/X]$ if X is a local link of P

admissible rule!

(E5) $P \equiv P' \Rightarrow P, Q \equiv P', Q$

(E6) $P \equiv P' \Rightarrow \{P\} \equiv \{P'\}$

implied by "congruence"

connectors

(E7) $X{=}X \equiv \mathbf{0}$

(E8) $X{=}Y \equiv Y{=}X$

admissible rule!

(E9) $X{=}Y,\ P \equiv P[Y/X]$

if P is an atom and X is a free link of P

(E10) $\{X{=}Y,\ P\} \equiv \{P\},\ X{=}Y$

if exactly one of X and Y is a free link of P

Reduction semantics

(R1) $$\frac{P \to P'}{P, Q \to P', Q}$$

(R2) $$\frac{P \to P'}{\{P\} \to \{P'\}}$$

(R3) $$\frac{Q \equiv P \quad P \to P' \quad P' \equiv Q'}{Q \to Q'}$$

structural

connectors

(R4) $$\{X{=}Y,\ P\} \to X{=}Y,\ \{P\}$$

if X and Y are free links of $(X{=}Y, P)$

(R5) $$X{=}Y,\ \{P\} \to \{X{=}Y,\ P\}$$

if X and Y are free links of P

(R6) $$T\theta, (T :\!-\ U) \to U\theta, (T :\!-\ U)$$

θ is to instantiate process & rule contexts and bundles.

Reduction semantics, notes

(R1) $$\frac{P \to P'}{P, Q \to P', Q}$$

(R2) $$\frac{P \to P'}{\{P\} \to \{P'\}}$$

(R3) $$\frac{Q \equiv P \quad P \to P' \quad P' \equiv Q'}{Q \to Q'}$$

structural

connectors

(R4) $\{X{=}Y,\ P\} \to X{=}Y,\ \{P\}$

if X and Y are free links of $(X$

(R4)(R5) could be "downgraded" to standard library

(R5) $X{=}Y,\ \{P\} \to \{X{=}Y,\ P\}$

if X and Y are free links of P

(R6) $T\theta, (T :- U) \to U\theta, (T :- U)$

θ is to instantiate process & rule contexts and bundles.

Reduction semantics

- Can `p(A,A)` be reduced using `p(X,Y) :- q(Y,X)` ?
  - The LHS of the rule can't be α-converted to `p(A,A)`
  - However, because `p(A,A)` is equivalent to `p(X,Y), X=A, Y=A` (`X`, `Y` fresh links), it can be reduced as:

```
  p(A,A)
≡ p(X,Y), X=A, Y=A
→ q(Y,X), X=A, Y=A
≡ q(A,A)
```

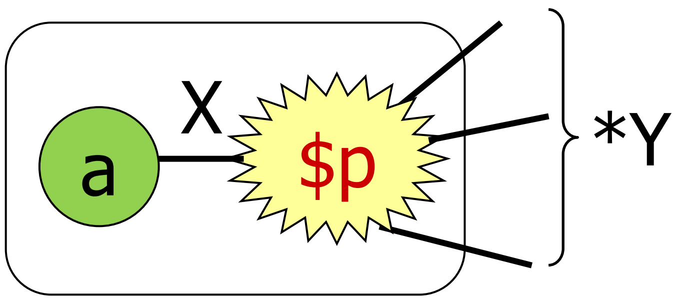
Process contexts

- *Process(or graph)-level variables* for :
  - migration (across membranes)
  - cloning and deletion (without tedious graph traversal)
- Two different families:
  1. *Untyped* — to capture “the rest of the nodes” (= “context”) of the enclosing membrane
  2. *Typed* — to capture a graph with a specific “shape” (= “type”), which is either
     a. pre-defined (int, unary, ground, …) or
     b. user-defined (by a grammar specified by “typedef”)

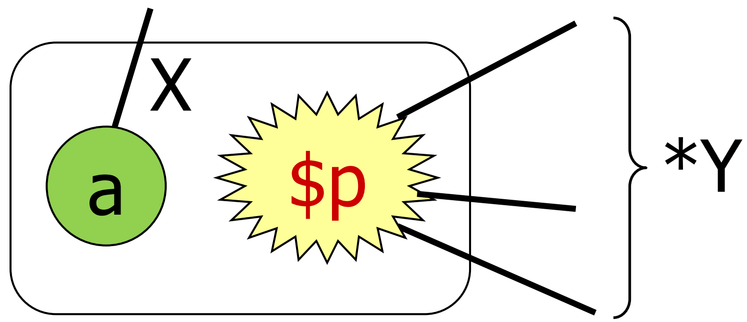
Representing the local context of a membrane

Free links matter!

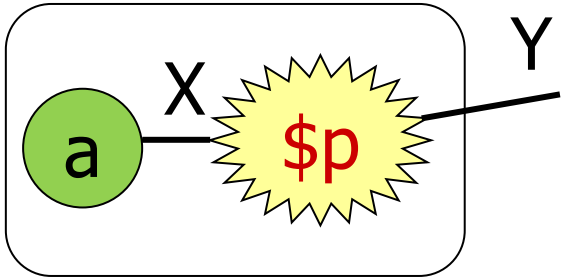
{a(X), $p[X|*Y]} :-



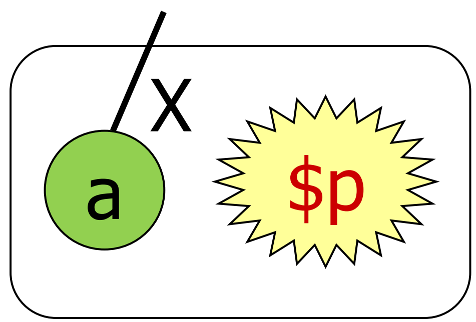
{a(X), $p[|*Y]} :-



{a(X), $p[X,Y]} :-

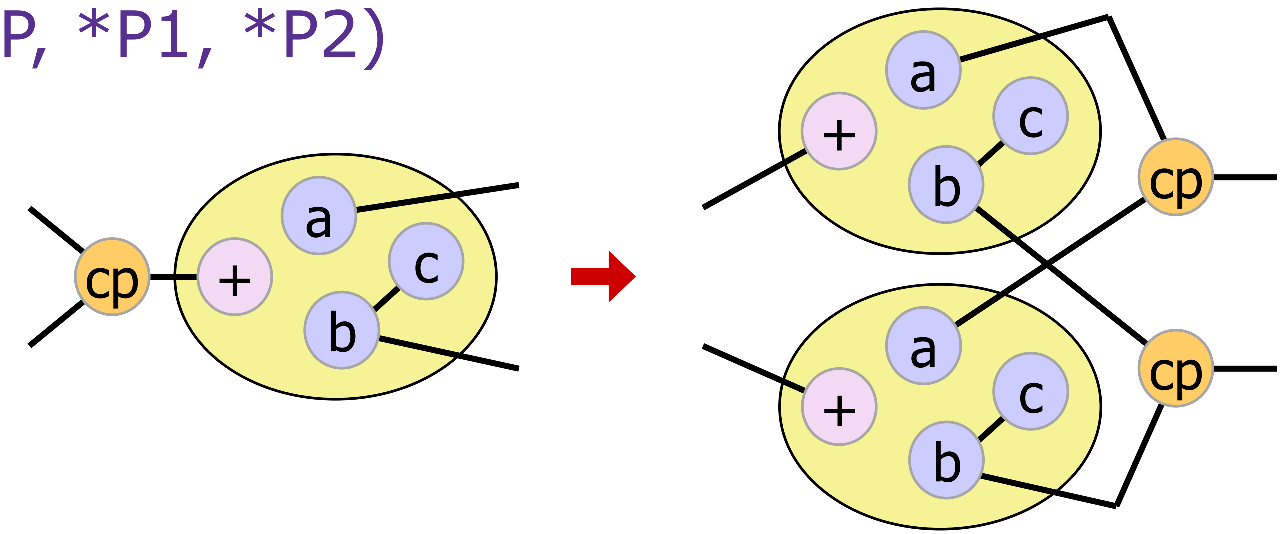


{a(X), $p[]} :-

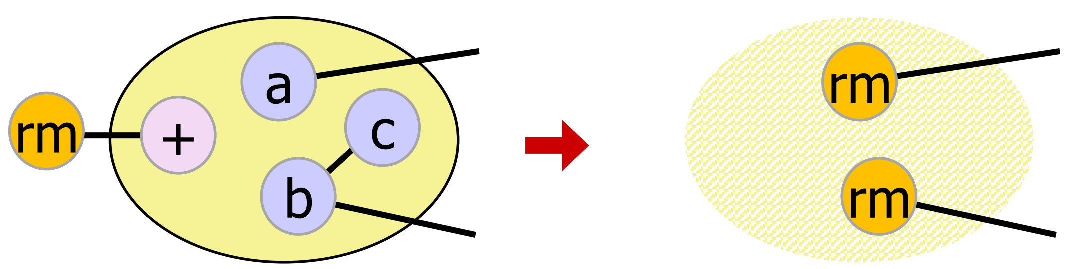


Cloning and deletion

- cp(S,S1,S2), {+S, $p[|*P]} :-
  {+S1, $p[|*P1]}, {+S2, $p[|*P2]},
  cp(*P, *P1, *P2)

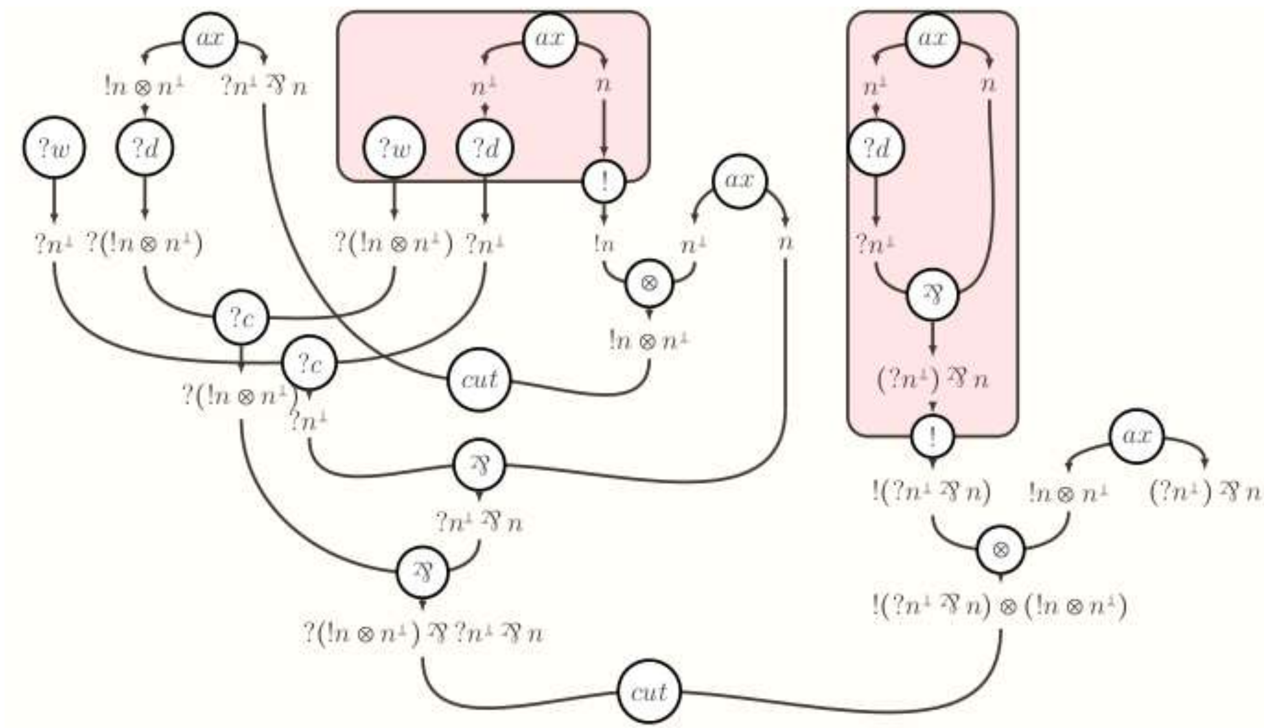


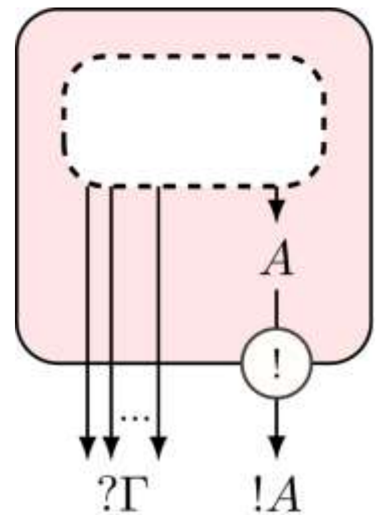
- rm(S), {+S, $a[|*X]]} :- rm(*X)



Affinity with Proof-Net Cut Elimination [APLAS 2023]

- MELL (Multiplicative Exponential Linear Logic) (Girard)
  = Multiplicative Linear Logic (MLL) + Exponential operators !, ?
- Proof Net = graphical representation of a sequent-calc. proof

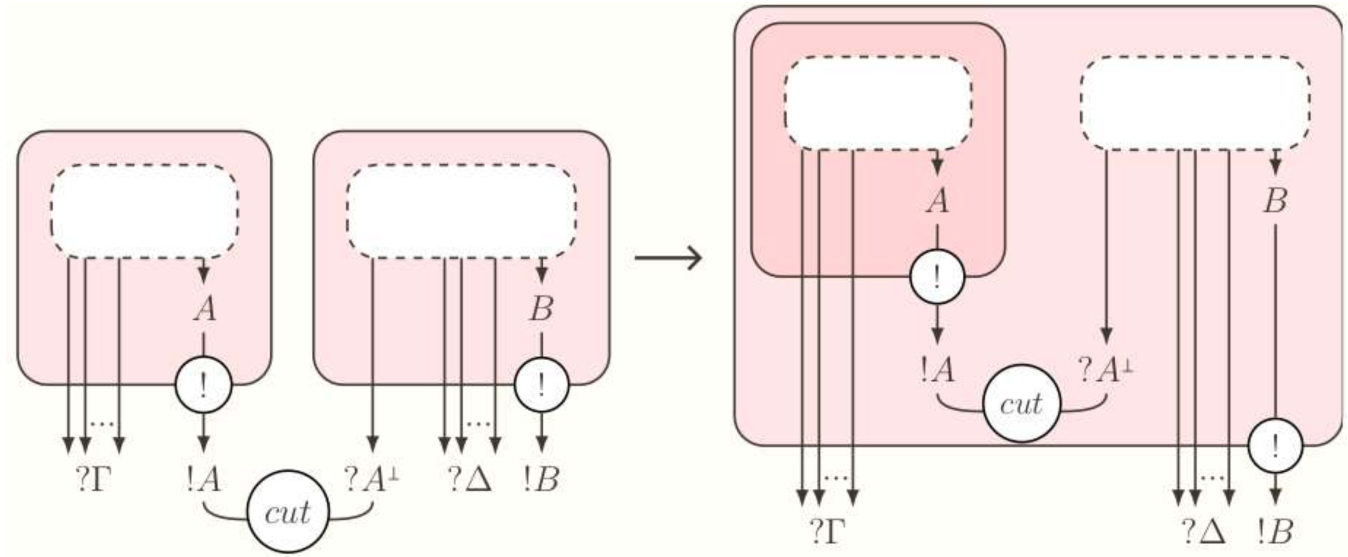


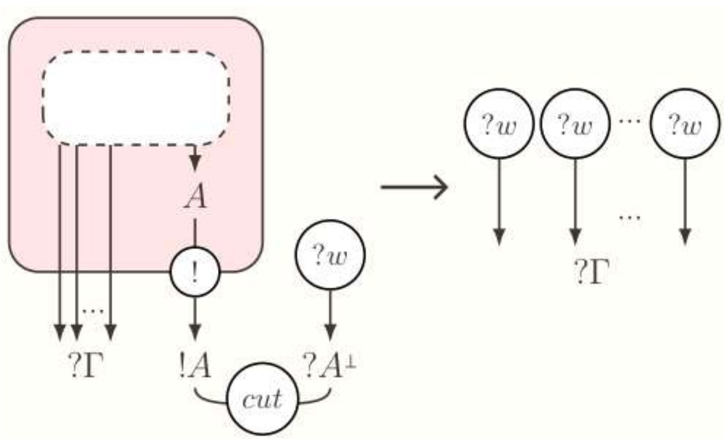


Promotion box for the !-rule
to protect the inner items

Affinity with Proof-Net Cut Elimination [APLAS 2023]

- Cut elimination translates to proof net reduction
- Examples:






```
cut_elimination_nested@@
{'!'(A,B),$p1[A|*X]},{$p2[C|*X]},cut{+(B),+(C)}
:- {{'!'(A,B),$p1[A|*X]},$p2[C|*X],cut{+(A),+(B)}}.
```

```
cut_elimination_weakening@@
{'!'(A,B),$p[A|*X]},'?w'(C), cut{+(B),+(C)}
:- nlmem.kill({trash(A),$p[A|*X]},'?w').
```

Typed process contexts

- Process-level variables (wildcards) matching graphs with specific "shape"
  - the shape is specified in a guard:

```
p(L), $n[L] :- int($n) | …
p($n) :- int($n) | …
```

type constraint

  - can be considered as *rule schemata*

connected graph with given 'roots'

- Type hierarchy:
  - int, float, string $\sqsubseteq$ unary $\sqsubseteq$ ground
  - '<' $\sqsubseteq$ int $\times$ int , '=' $\sqsubseteq$ ground $\times$ ground
  - CSLMNtal supports typedef (CFG-based; cf. type graphs)

Negative application conditions (NACs)

- As a *concurrent* language based on *subgraph* matching of *open* graphs, NACs (non-existence or non-progress) are supported in connection with membranes and/or contexts
- Comes in various forms:
  - Stability flag for termination detection `{ ... }/ :- ...`
  - Membrane without a context (e.g., {a(X),b(Y)})
  - Checking of a context (e.g., \$p ≠ 0) (e.g., \$p ≠ (a,\$q))
  - (to be supported in a general form as one of the quantifiers [LOPSTR 2024])

Natural numbers in axiomatic set theory

- Membrane computing supported as the other extreme

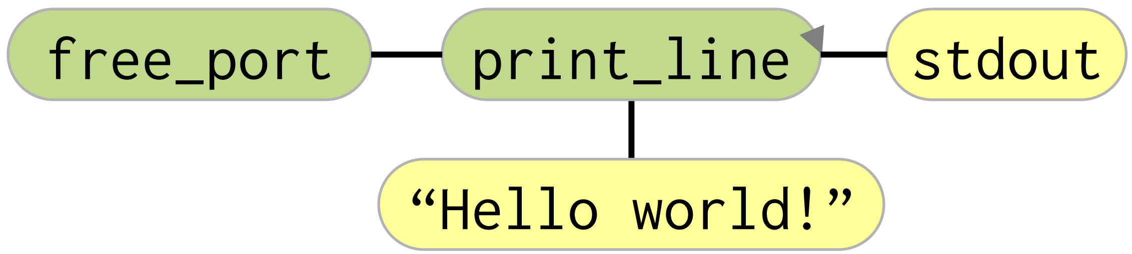
$$0 = \emptyset$$
$$1 = \{0\} = \{\emptyset\}$$
$$2 = \{0,1\} = \{\emptyset, \{\emptyset\}\}$$
$$3 = \{0,1,2\} = \{\emptyset, \{\emptyset\}, \{\emptyset, \{\emptyset\}\}\}$$

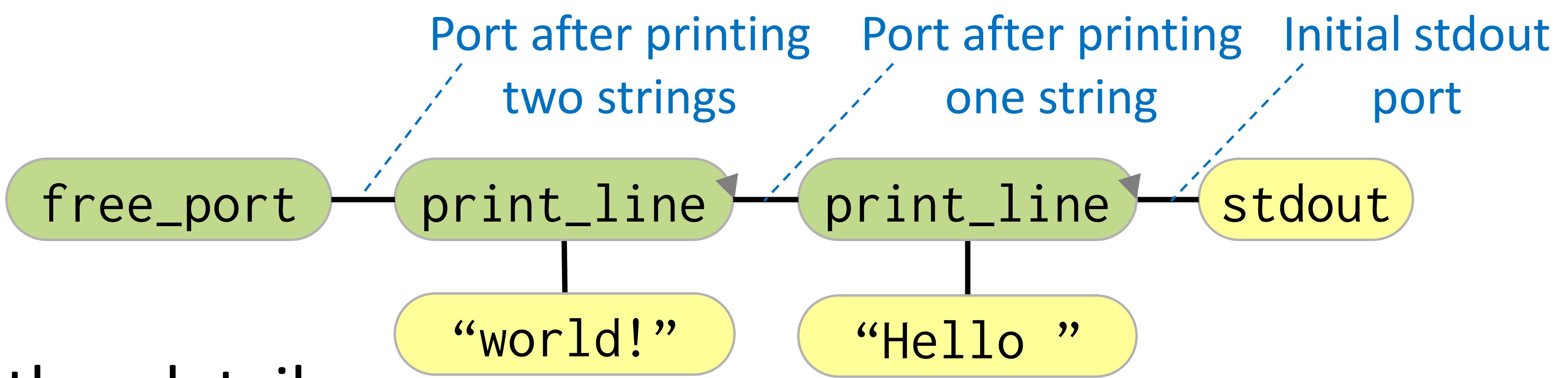
```
  succ, {$p[]} :- {$p[], {$p[]}}.

  succ, succ, succ, {}.
→ succ. succ. {{}}.
→ succ. {{{}}, {}}.
→ {{{}, {{}}}, {}, {{}}}.
```

Hello world! — I/O in declarative style

- Stream-based (or monadic) I/O

```
io.print_line(io.stdout,“Hello world!",io.free_port).
```



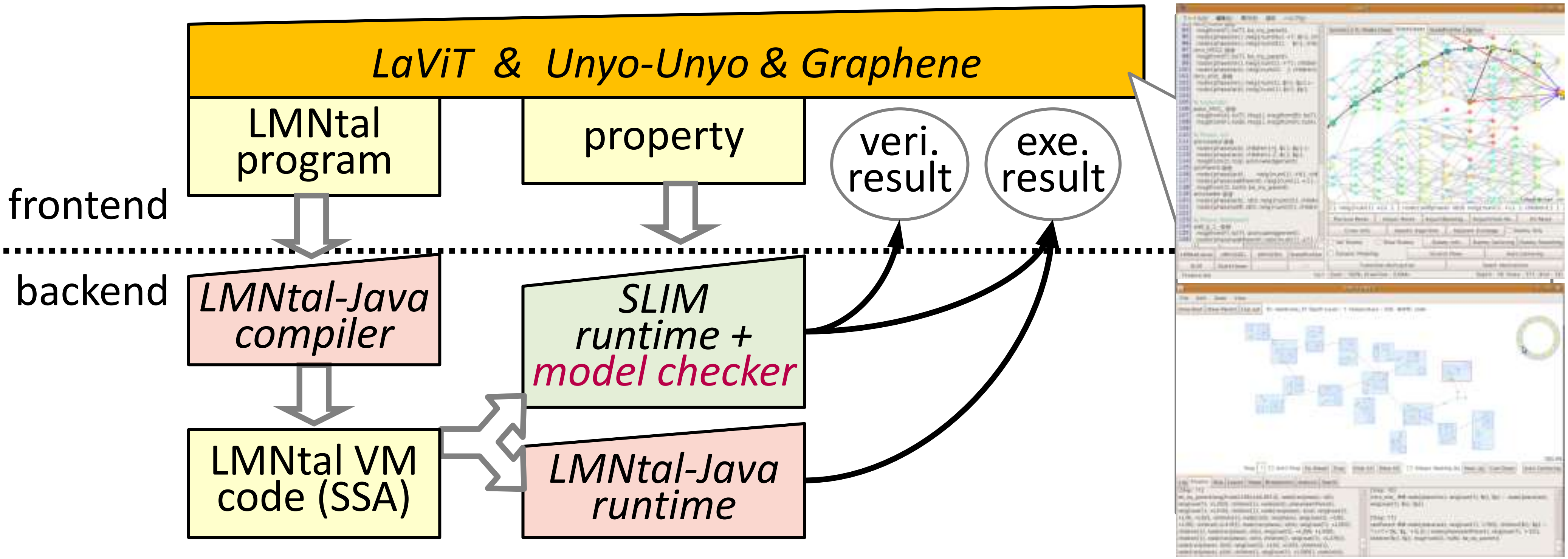


- Further details:

  https://www.ueda.info.waseda.ac.jp/lmntal/index.php?Library%20Reference#io

Implementation overview

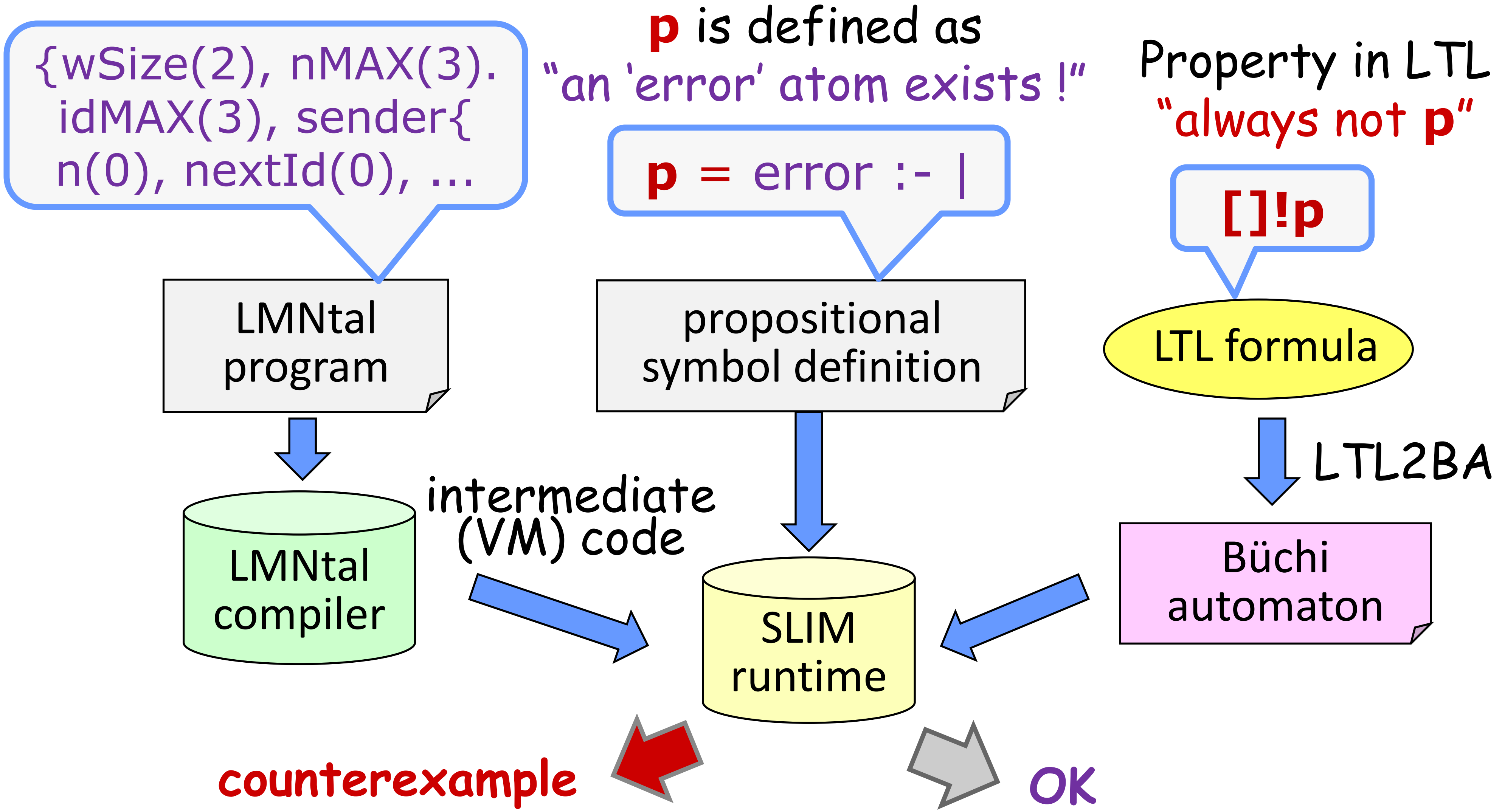
LMNtal-Java	2002–	Java	30kLOC	compiler + runtime w/FLI
Unyo-Unyo	2006–	Java	16kLOC	execution visualizer
SLIM	2007–	C++	34kLOC	faster runtime w/model checker
LaViT	2008–	Java	27kLOC	IDE w/state-space visualizer
Graphene	2014–	Scala	2kLOC	2nd-generation visualizer



Model checking in LMNtal: Motivations

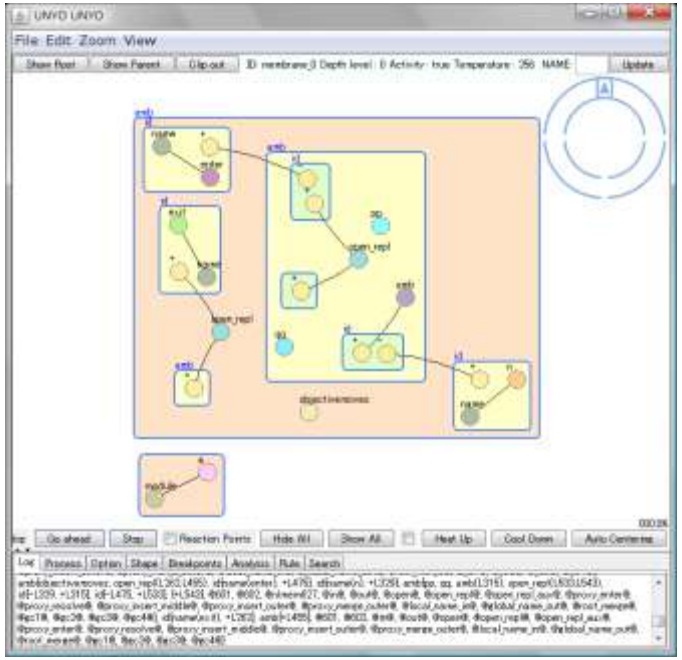
- LMNtal is good at modeling systems which computer-aided verification is concerned with, including
  - state transition systems (automata) and
  - concurrent systems.
- LMNtal is at the same time a full-fledged programming language allowing infinite states.
  - No gap between modeling and programming languages (cf. SPIN, nuSMV, . . .)
  - As a fine-grained concurrent language, supporting verification is highly desirable
- Why not build an integrated development and verification environment?

LMNtal model checker

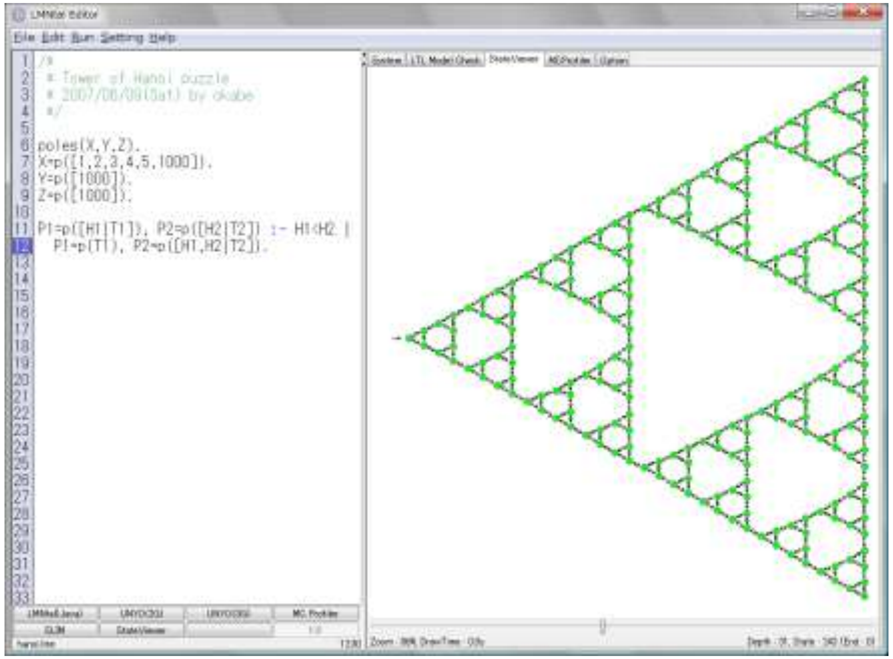


Model checking in LMNtal: Strengths

- LaViT supports the understanding of models with and without errors, not just bug catching
  - workbench for designing and analyzing models
  - complementary to fast, black-box checkers
- Hierarchical graphs feature built-in *symmetry reduction*



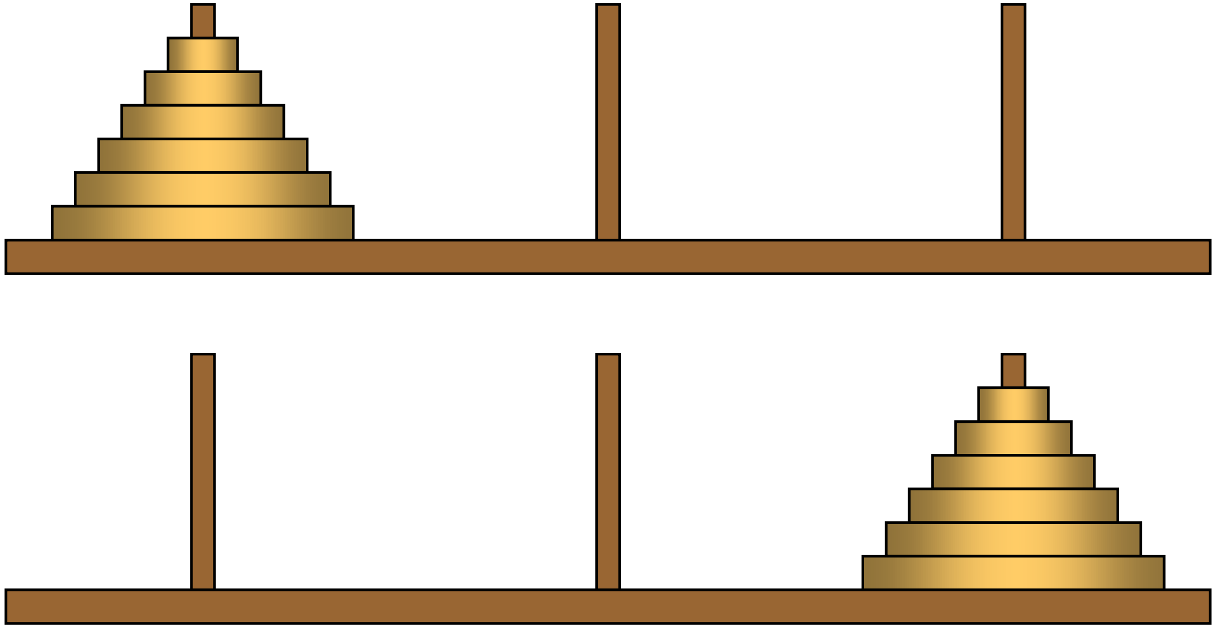
Unyo-Unyo Visualizer



StateViewer (Tower of Hanoi)

Demo: The tower of Hanoi

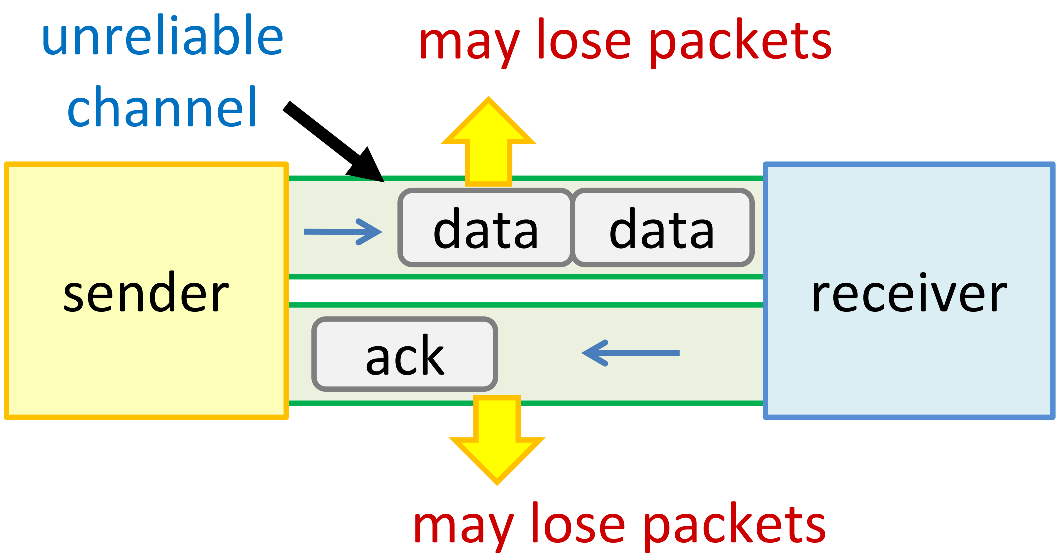
```
poles(p([1,2,3,4,5,6,99]),p([99]),p([99])).
```



```
P1=p([$h1|$t1]), P2=p([$h2|$t2]) :- $h1<$t2 |
    P1=p(T1), P2=p([$h1,$h2|$t2]).
```

Demo: Sliding window protocol (SWP)

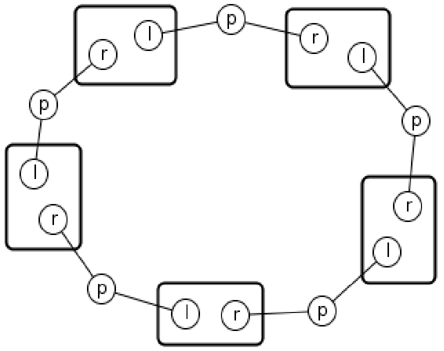
- SWP: transmission protocol used in TCP
  - Sends data packets (up to window size) without waiting for acknowledgment
    - Rollbacks if some item seems lost
  - Channels may lose data and acks



$\Box(\text{send} \Rightarrow \Diamond \text{ack})$?

Coping with heavy structure and state explosion

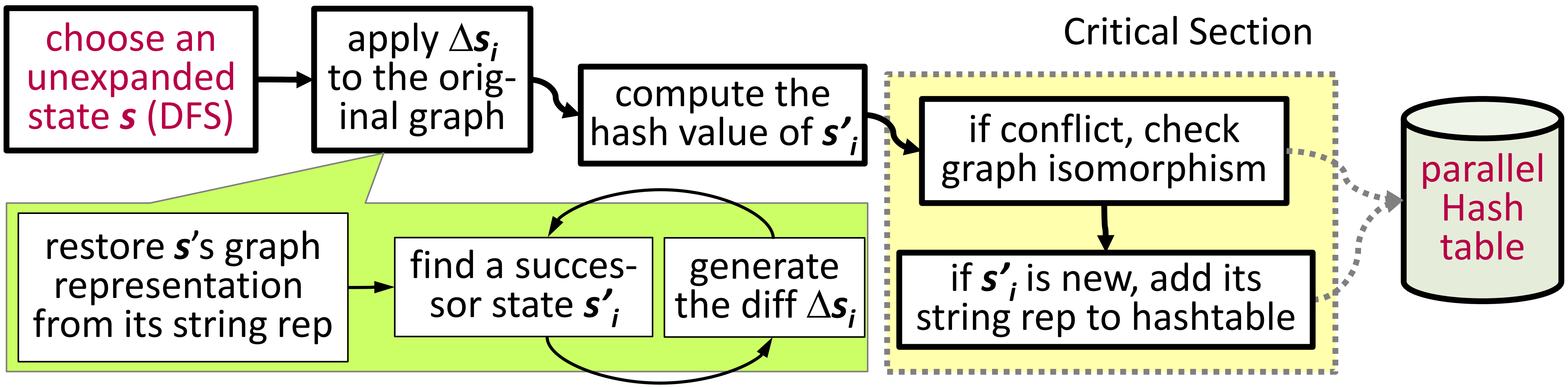
- Managing the state space of graphs requires both *space-efficient* graph representation and *time-efficient* isomorphism checking. They are supported by:
  - Hashing with parallel hash-table
  - State encoding (serialization)
    - Non-canonical encoding
    - Canonical encoding (labelling)
  - Tree compression
  - Backward execution
  - Parallel state-space search
  - Partial-order reduction



Original	>2KB
Encoded	70B

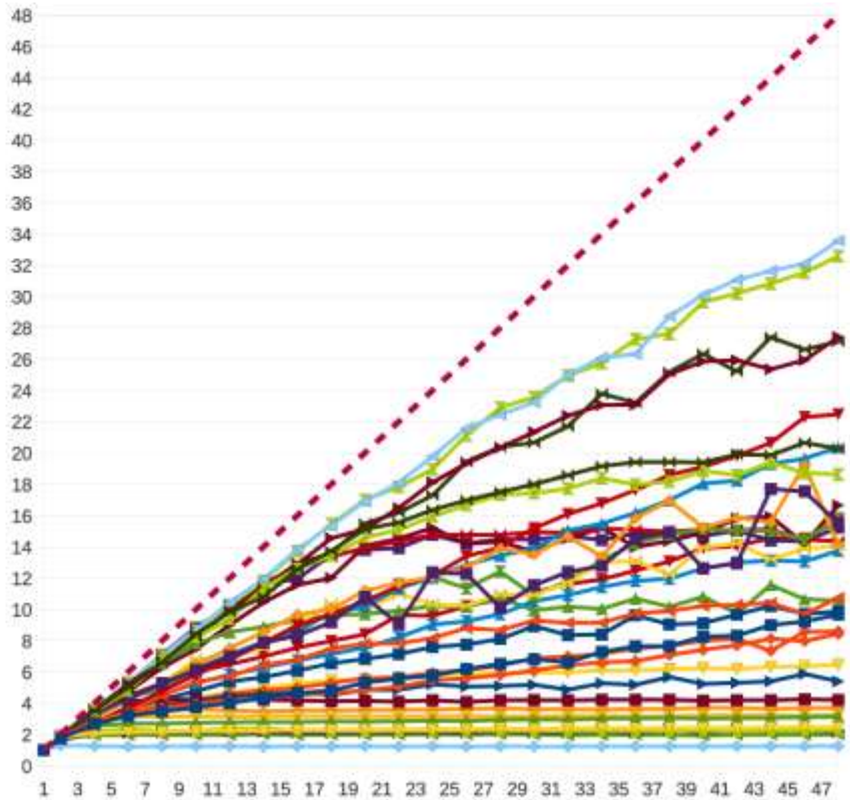
Parallelizing the LMNtal model checker

- Construct a state-space graph by *stack-slicing parallel DFS*
- Apply an DiVinE-like algorithm to search a counterexample
- Built by (i) analyzing the sequential model checker (25kLOC), (ii) ensuring thread safety, and (iii) improving scalability on many-core processors
  - dynamic load balancing, parallel hash table
  - introducing parallel memory allocator

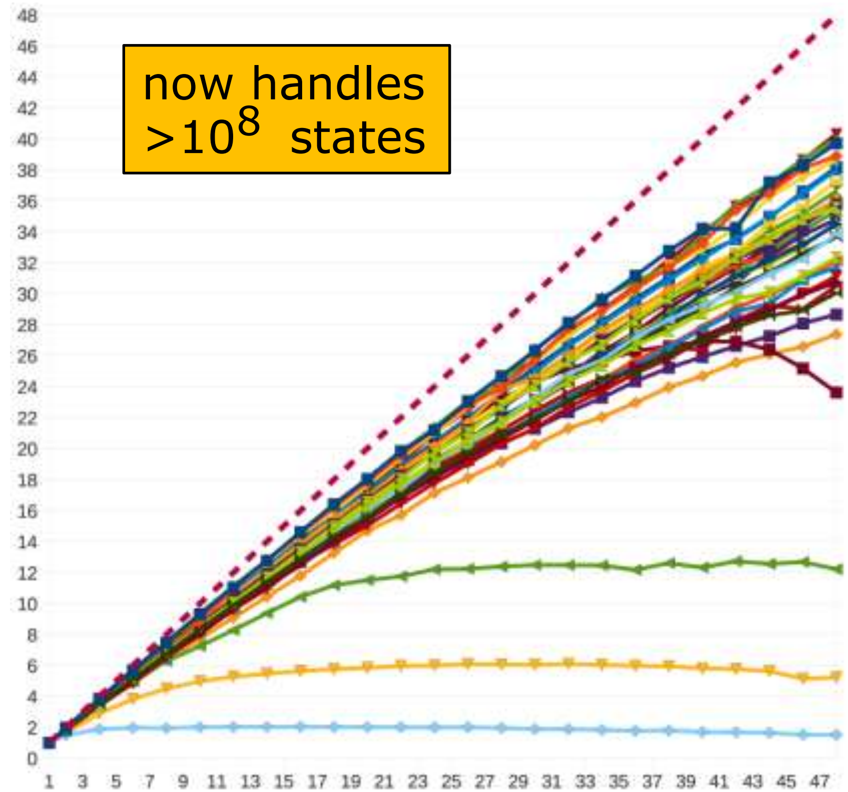


* M. Gocho, T. Hori and K. Ueda, *Computer Software* **28**(4), pp.137-157, 2011

Speedup of the LMNtal parallel model checker

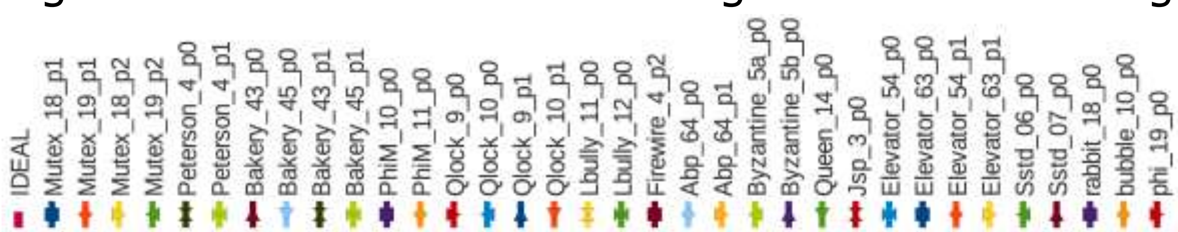


stack slicing



stack slicing with work stealing

AMD Opteron (2.3GHz) 12-core x 4, 256GB of memory

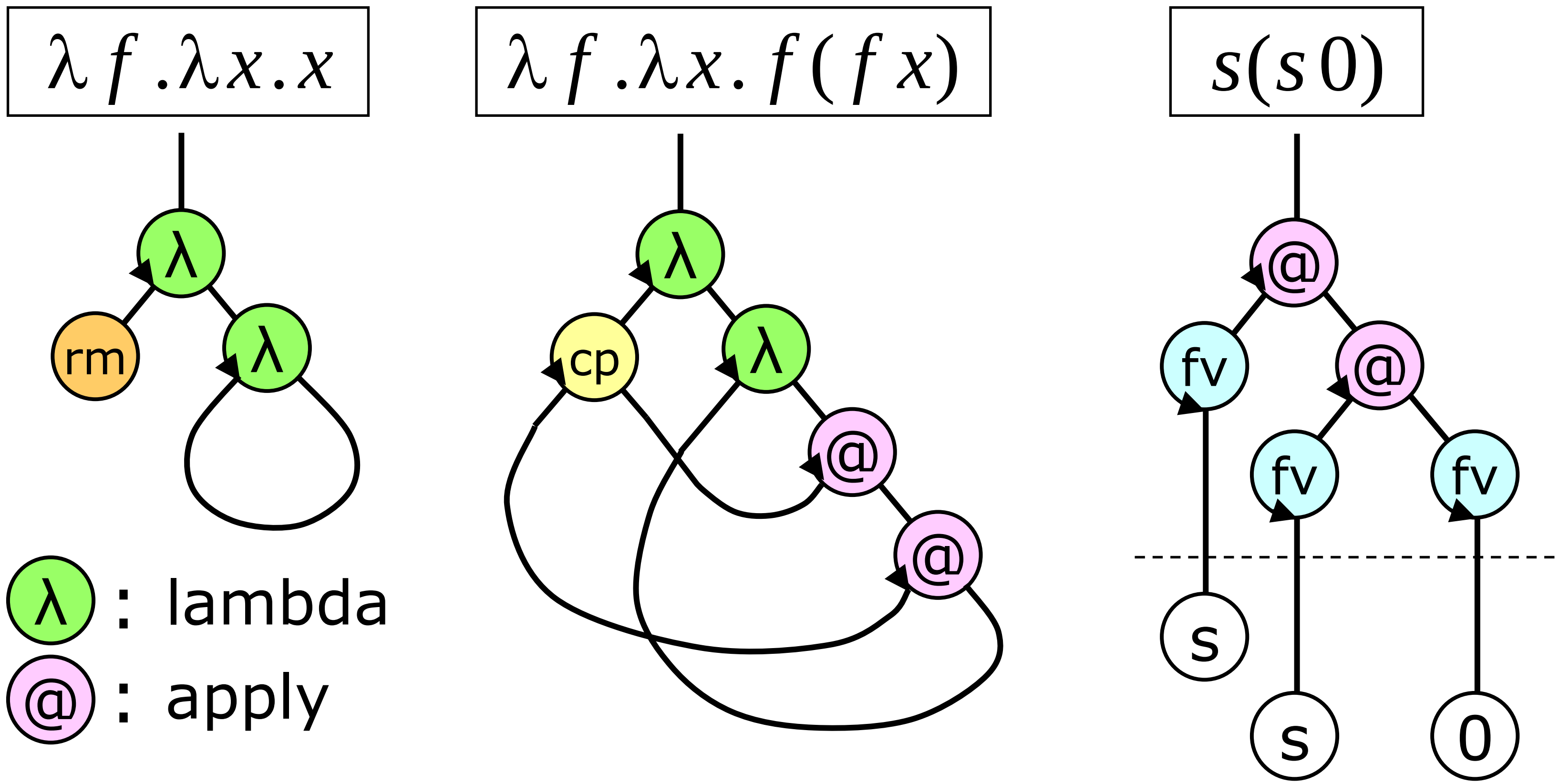


M. Gocho, T. Hori and K. Ueda, *Computer Software*, **28**(4), 2011

Extensions (almost orthogonal to each other)

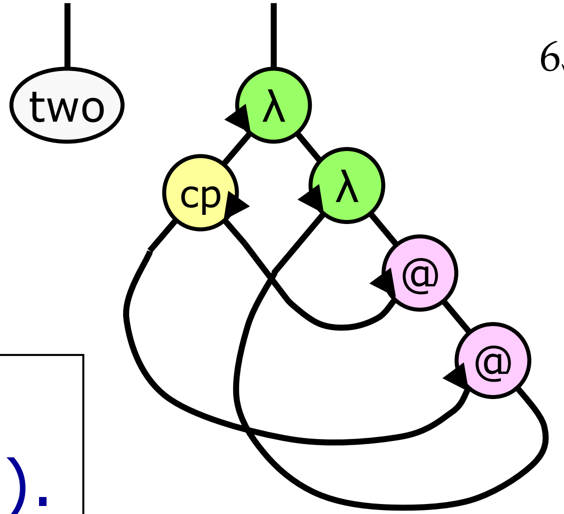
- Hyperlinks (➔ HyperLMNtal)
- Contexts (wildcards)
  - HyperLMNtal (flexible handling of hyperlinks)
  - CSLMNtal (grammar-based context types)
- Rules for
  - “zero-step” (instantaneous) transition
  - “one-shot” use
  - meta-programming (“first-class” rules)
- Modules and foreign language interface
- Quantifiers [LOPSTR 2024]
- Static typing [PPDP 2024, LNCS 8865, etc.]

Lambda calculus, fine-grained [RTA 2008]



Unique up to cp+rm's equational theory
(= ACU : associativity & commutativity with unit)

Church Numeral Exponentiation



- Church numeral 2: $\lambda f.\lambda x.f(f x)$

```
lambda(cp(F0, F1),
    lambda(X, apply(F0, apply(F1, X))), N).
```

- 3^2 : $(((\lambda m.\lambda n.n\ m)\ 3)\ 2)$

```
N = two  :-
     N = lambda(cp(F0, F1),
              lambda(X, apply(F0, apply(F1, X)))).
N = three  :-
     N = lambda(cp(F0, cp(F1, F2)),
                lambda(X, apply(F0, apply(F1, apply(F2, X))))).
res = apply(apply(apply(two, three), fv(succ)), fv(0)).
H = apply(fv(succ), fv($i)) :- int($i) | H=fv($i+1).
```

The Encoding (1/2)

graph copying

β-reduction

```
✓ H=apply(lambda(A,B), C)  :-  H=B, A=C.

✓ lambda(A,B)=cp(C,D,L), {+L,$q}  :-
      C=lambda(E,F), D=lambda(G,H),
      A=cp(E,G,L1), B=cp(F,H,L2),
      {{+L1},+L2,sub(S)}, {super(S),$q}.

✓ apply(A,B)=cp(C,D,L), {+L,$q}  :-
      C=apply(E,F), D=apply(G,H), A=cp(E,G,L1), B=cp(F,H,L2),
      {+L1,+L2,$q}.

✓ cp(A,B,L1)=cp(C,D,L2), {{+L1,$p},+L2,$q}  :-
      A=C, B=D, {{$p},$q}.

✓ cp(A,B,L1)=cp(C,D,L2), {{+L1,$p},$q}, {+L2,top,$r}  :-
      C=cp(E,F,L3), D=cp(G,H,L4), {{+L3,+L4,$p},$q},
      A=cp(E,G,L5), B=cp(F,H,L6), {+L5,+L6,top,$r}.

✓ fv($u)=cp(A,B,L), {+L,$q}  :-  unary($u) |
      A=fv($u), B=fv($u), {$q}.
```

The Encoding (2/2)

graph destruction

```
lambda(A,B)=rm  :-  A=rm, B=rm.
apply(A,B)=rm  :-  A=rm, B=rm.
cp(A,B,L)=rm, {+L,$q}  :-  A=rm, B=rm, {$q}.
✓ cp(A,B,L)=rm, {{+L,$p},$q}  :-  A=rm, B=rm, {{$p},$q}.
A=cp(B,rm,L), {+L,$p}  :-  A=B, {$p}.
A=cp(rm,B,L), {+L,$p}  :-  A=B, {$p}.
rm=rm  :-  .
fv($u)=rm  :-  unary($u) | .
```

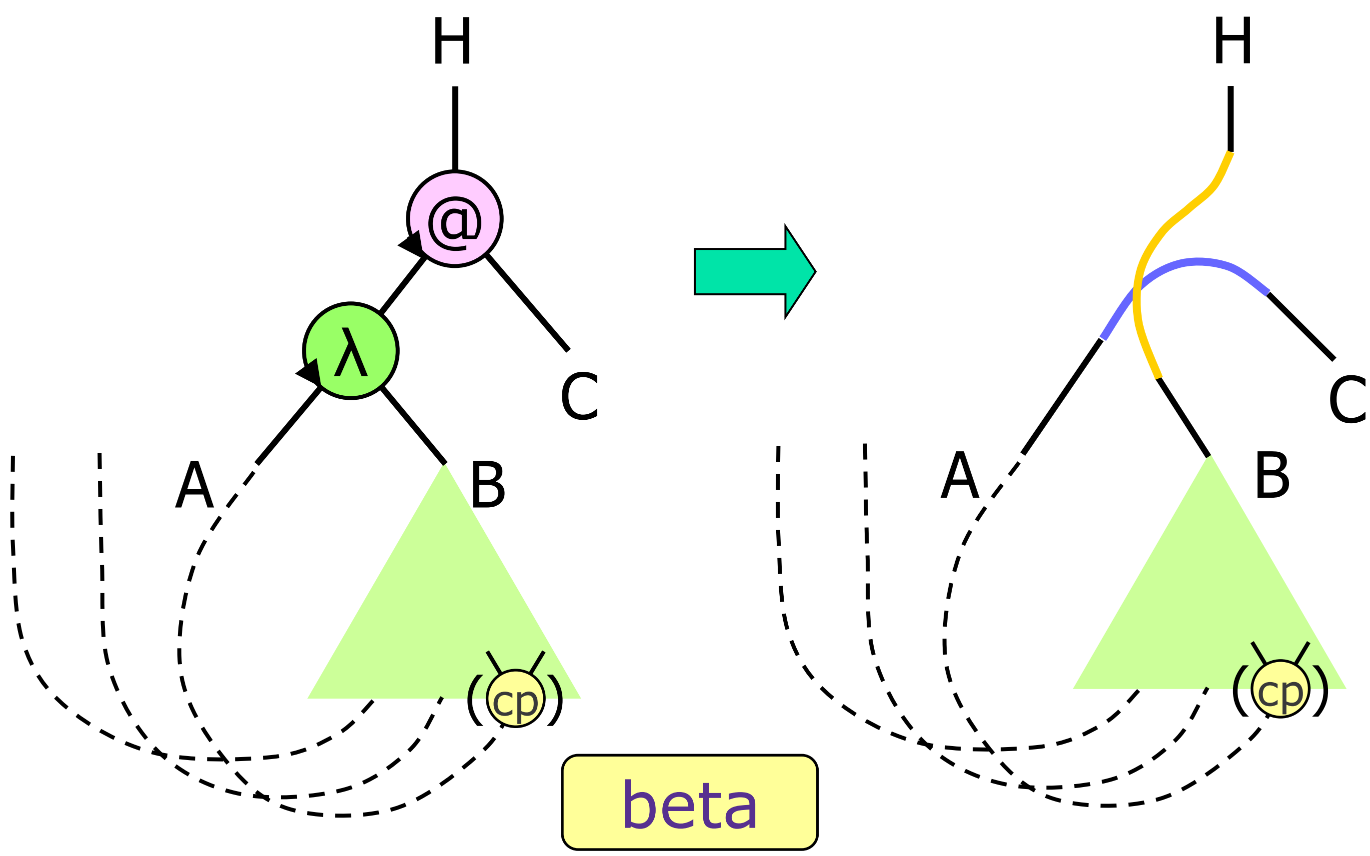
color management

```
✓ {{},$p,sub(S)}, {$q,super(S)}  :-  {$p,$q}.
A=cp(B,C)  :-  A=cp(B,C,L), {+L,top}.
{top}  :-  .
```

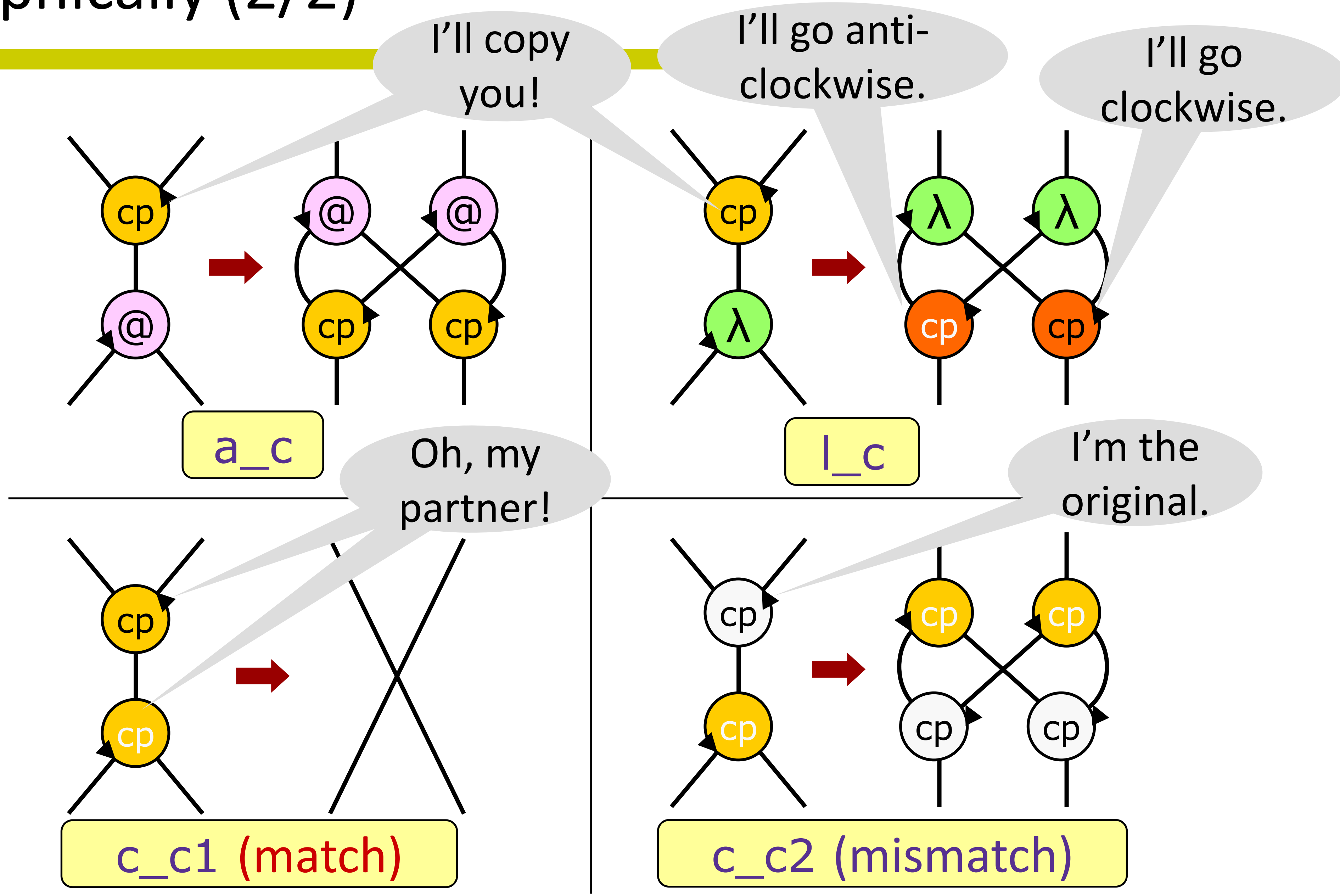
✓: Eight essential rules
(the other rules are for tidying up and initialization)

Graphically (1/2)

H=apply(lambda(A,B), C) :- H=B, A=C.



Graphically (2/2)

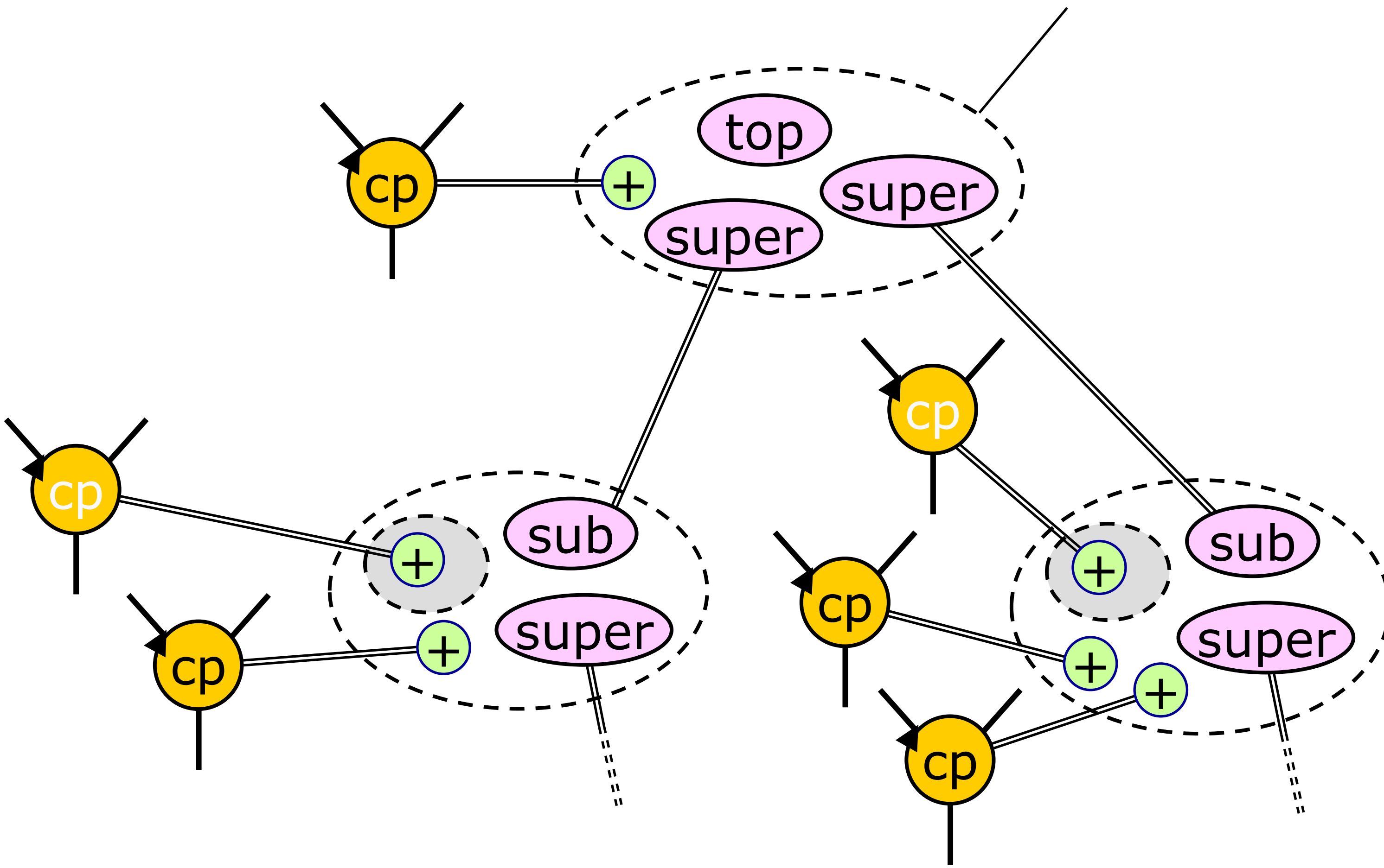


The Key Idea

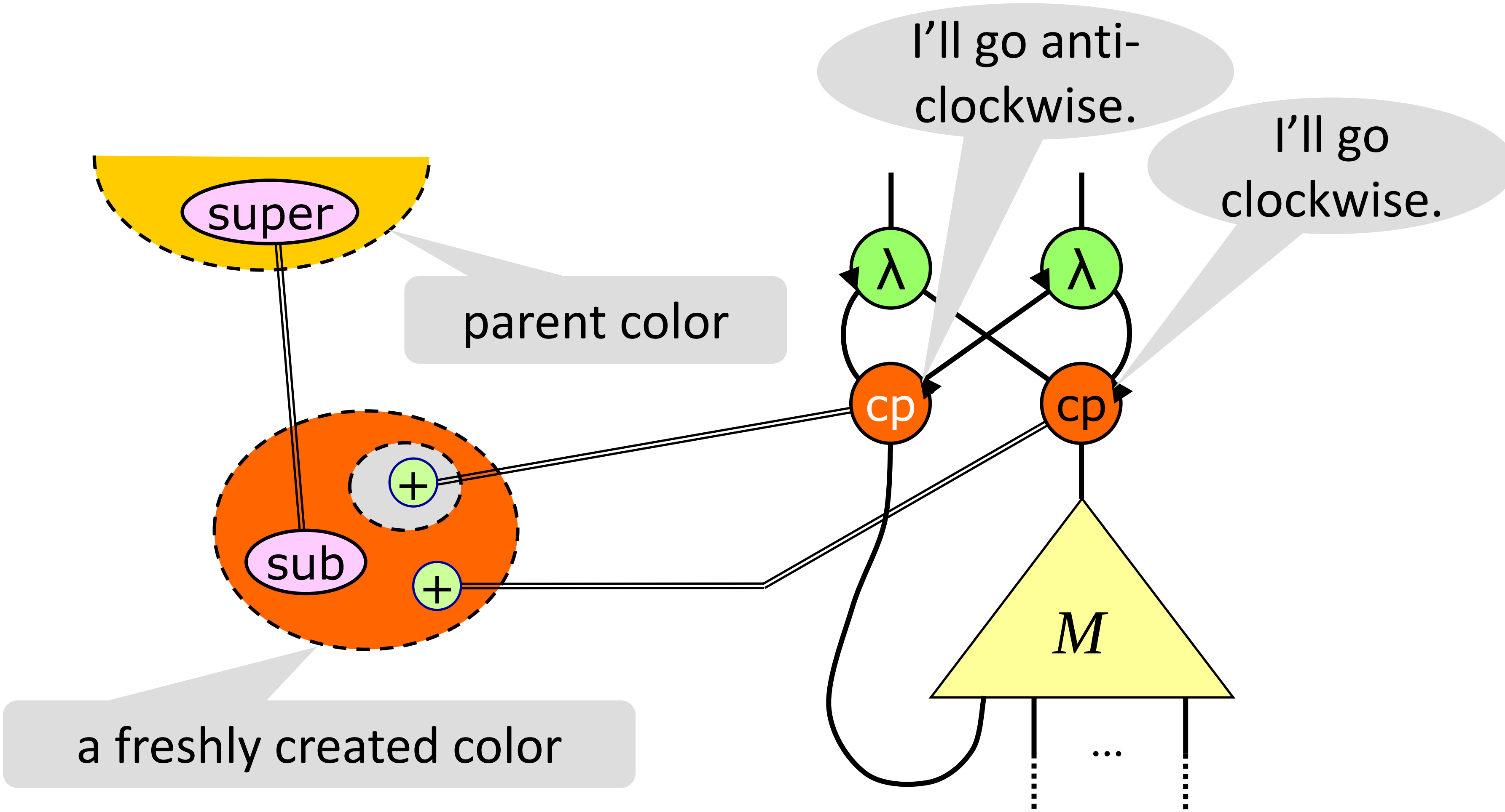
- Which of $\boxed{c\_c1}$ and $\boxed{c\_c2}$ to apply?
- Existing methods used two colors or natural numbers to label **cp**'s
- We employ hierarchical colors (= **local names**)
  - whenever a **cp** encounters a λ, two **cp**'s are created to copy the abstraction.
  - when all the new **cp**'s running anti-clockwise hit their counterparts and disappear, the remaining **cp**'s become **cp**'s.

Color (= local name) Management

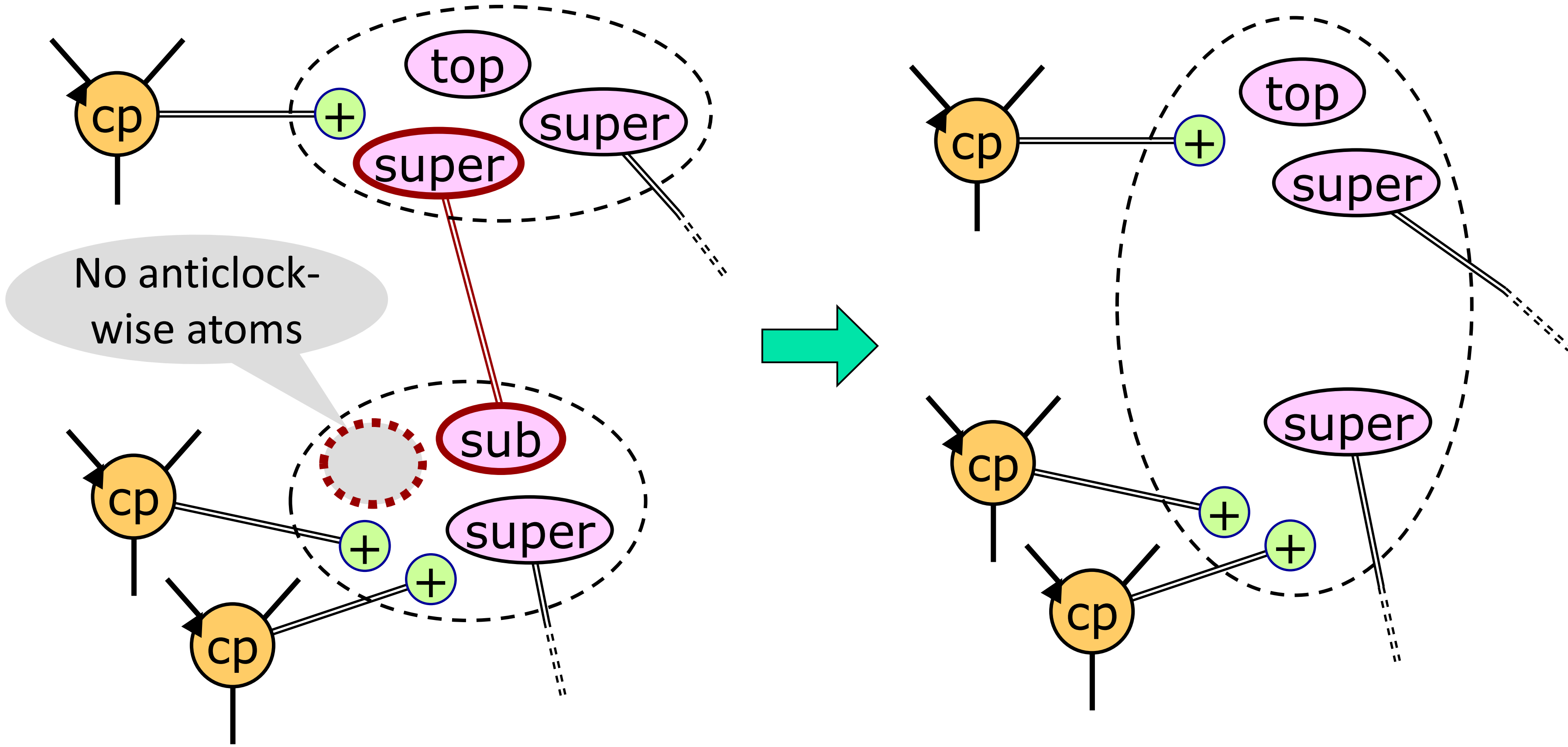
- Colors are encoded using membranes.



The RHS of the l_c rule



Promotion (Color Fusion) (cf. “retirement”)



Good news: Promotion need not be instantaneous; can be delayed safely.

Lambda calculus, coarse-grained [IEEE Access 2021]

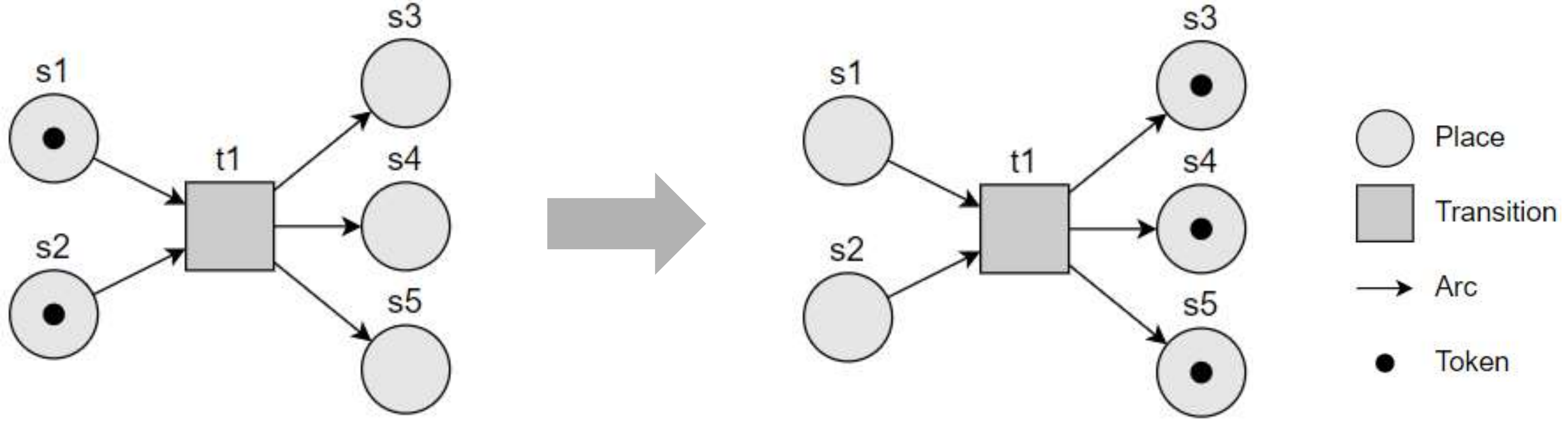
- One-to-one translation of the textbook definition, where 'ground' for hypergraphs follows ordinary links, then copies/shares hyperlinks depending on how they occur

```
beta@@ R=app(lam(X,A),B) :- R=subst(A,X,B).
var1@@ R=subst($x,$x,N) :- hlink($x) | R=N.
var2@@ R=subs($x,$y,$n) :-
         $x \=$y, ground($n,1) | R=$x.
abs@@  R=subst(lam($x,M),Y,N):-
         R=lam($x,subst(M,Y,N)).
app@@  R=subst(app(M1,M2),$x,$n) :-
         hlink($x), ground($n,1) |
         R=app(subst(M1,$x,$n),
               subst(M2,$x,$n)).
```

$(\lambda x. A)B$
$\to A[x \mapsto B]$

Substitution with no precaution on variable capture

Petri Nets [LOPSTR 2024]

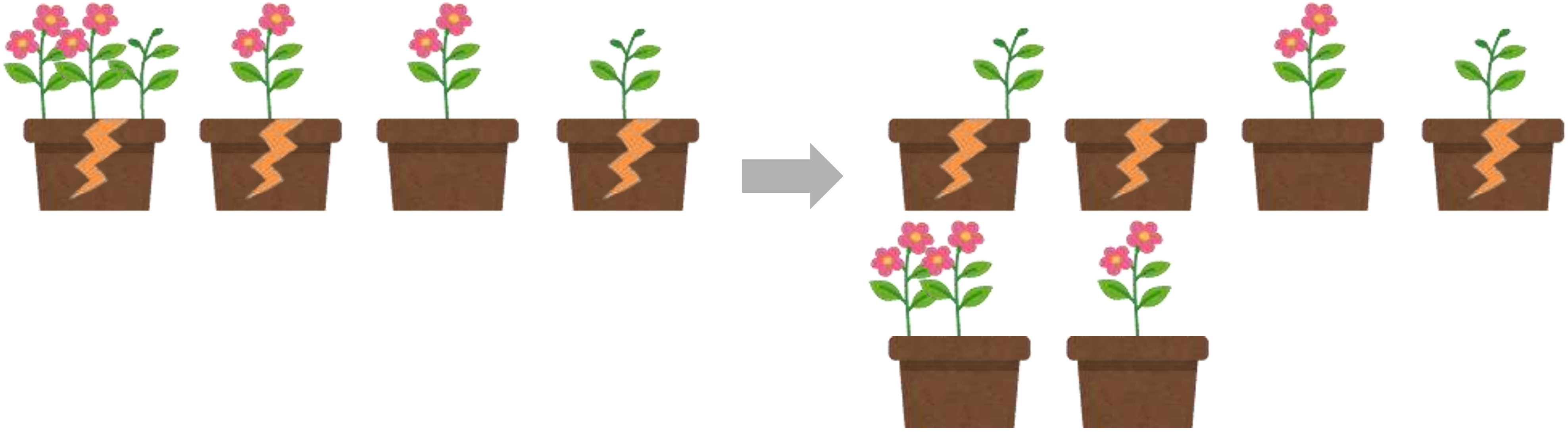


Labelled $\forall^{>0}$ quantifier

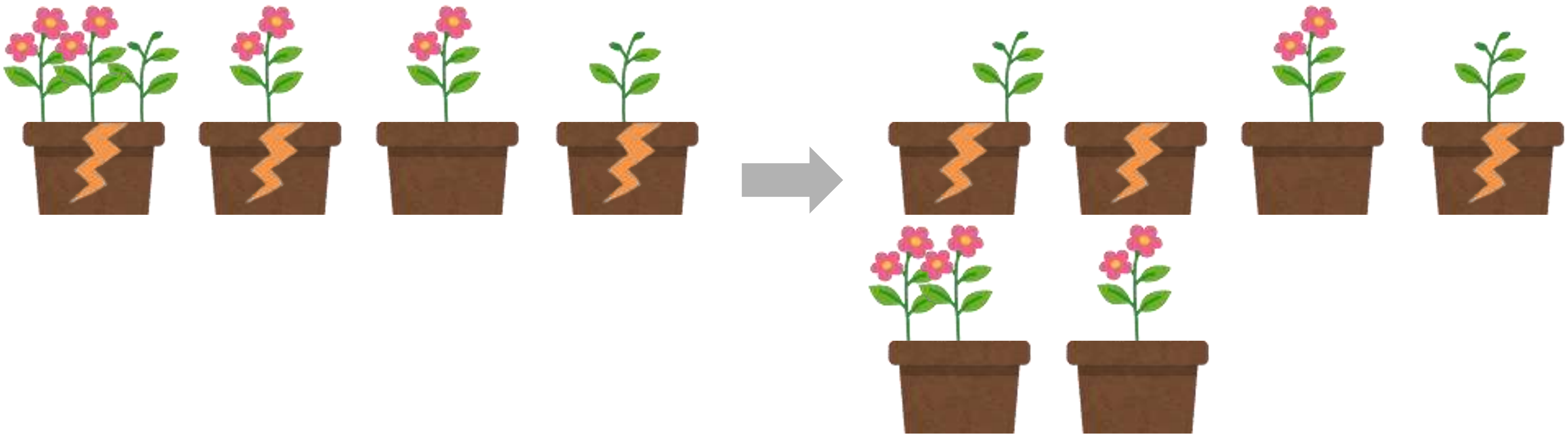
```
{token,-X1},{token,-X2}, {+Y1},{+Y2},{+Y3}.    //places
{+X1,+X2,-Y1,-Y2,-Y3}.                          //transition

M<+>{token,-X,$p1}, {M<+> +X,N<+> -Y}, N<+>{+Y,$p2} :-
M<+>{-X,$p1}, {M<+> +X,N<+> -Y}, N<+>{token,+Y,$p2}.
```

Repotting the Geraniums [LOPSTR 2024]

- “There are several pots, each with several geranium plants. Some pots were broken because the geraniums filled the space with their roots. New pots are prepared for the broken pots with flowering geraniums and all the flowering geraniums are moved to the new pots.” [Rensink et al. 2009]



Repotting the Geraniums [LOPSTR 2024]



```
{cracked, flowering, flowering, unflowering},
{cracked, flowering}, {uncracked, flowering},
{cracked, unflowering}.
```

```
M<+>{cracked, N<+>flowering, $p} :-
M<+>({cracked, $p}, {uncracked, N<+>flowering}).
```

Experiences

- Generalized data structures with more and less edges
- Concurrency with controllable granularity
- Graph-based model checking (up to 10^9 states)
  - with many implementation techniques
- Unified framework of computation and verification
- Visualization for understanding (cf. verifying) systems

Implementation (available from GitHub) >99% done by students joining and graduating every year

Thank you for your attention!

Questions/suggestions welcome, e.g.,

"Can LMNtal express and execute X ?"
"Why don't you add feature Y ?"
"Could you help me encode my idea Z ?"

To try yourself, visit
http://www.ueda.info.waseda.ac.jp/lmntal/ and choose LaViT