Logic Programming

Logic programs: a set of Horn-clauses

H_1 :- G_{11}, G_{12}, ...G_{1m}.
H_1 :- G_{21}, G_{22}... G_{2n}.
H_1.
H_2:-...
...

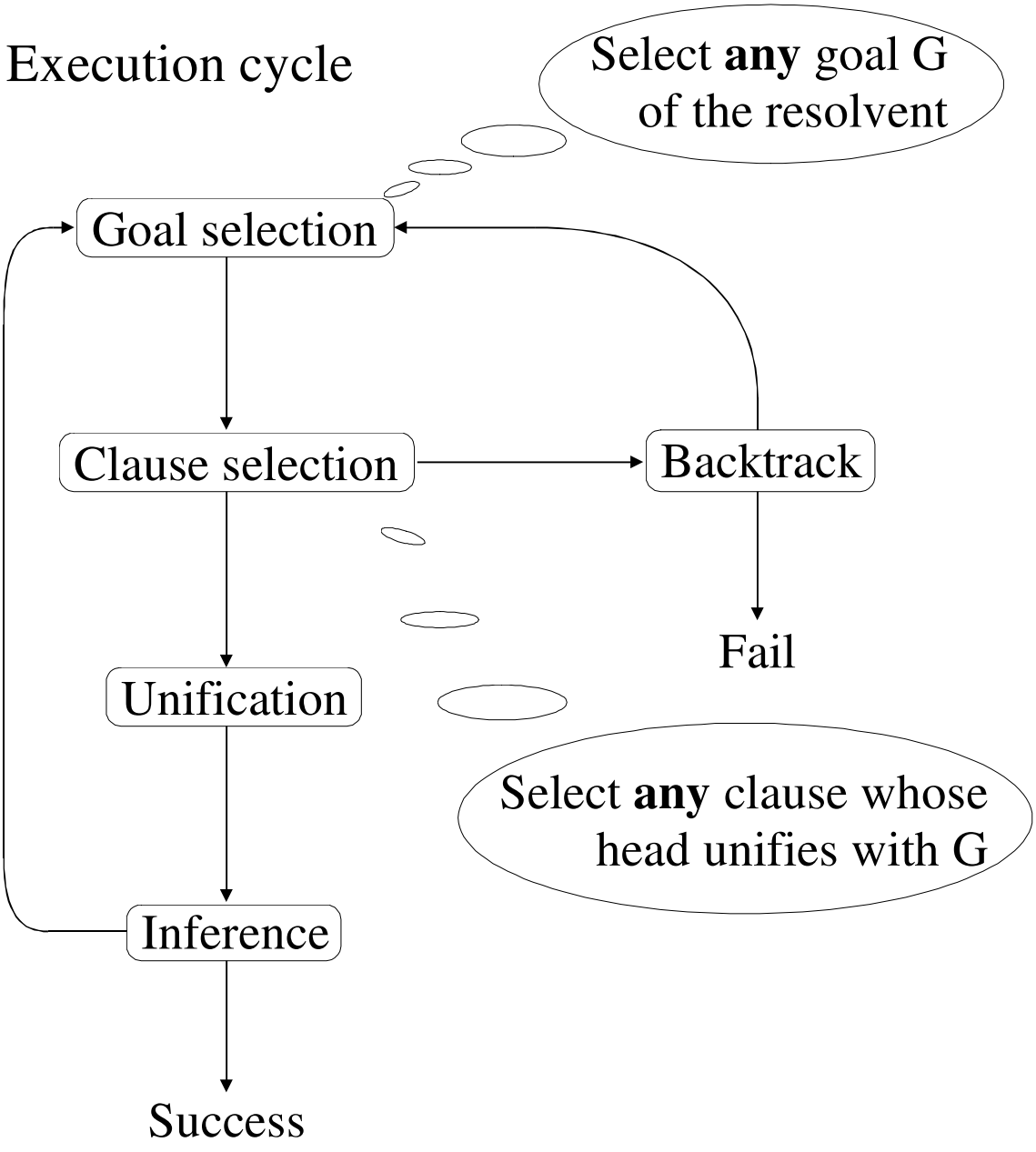
and a query

:- Q_1, Q_2, ...Q_k.

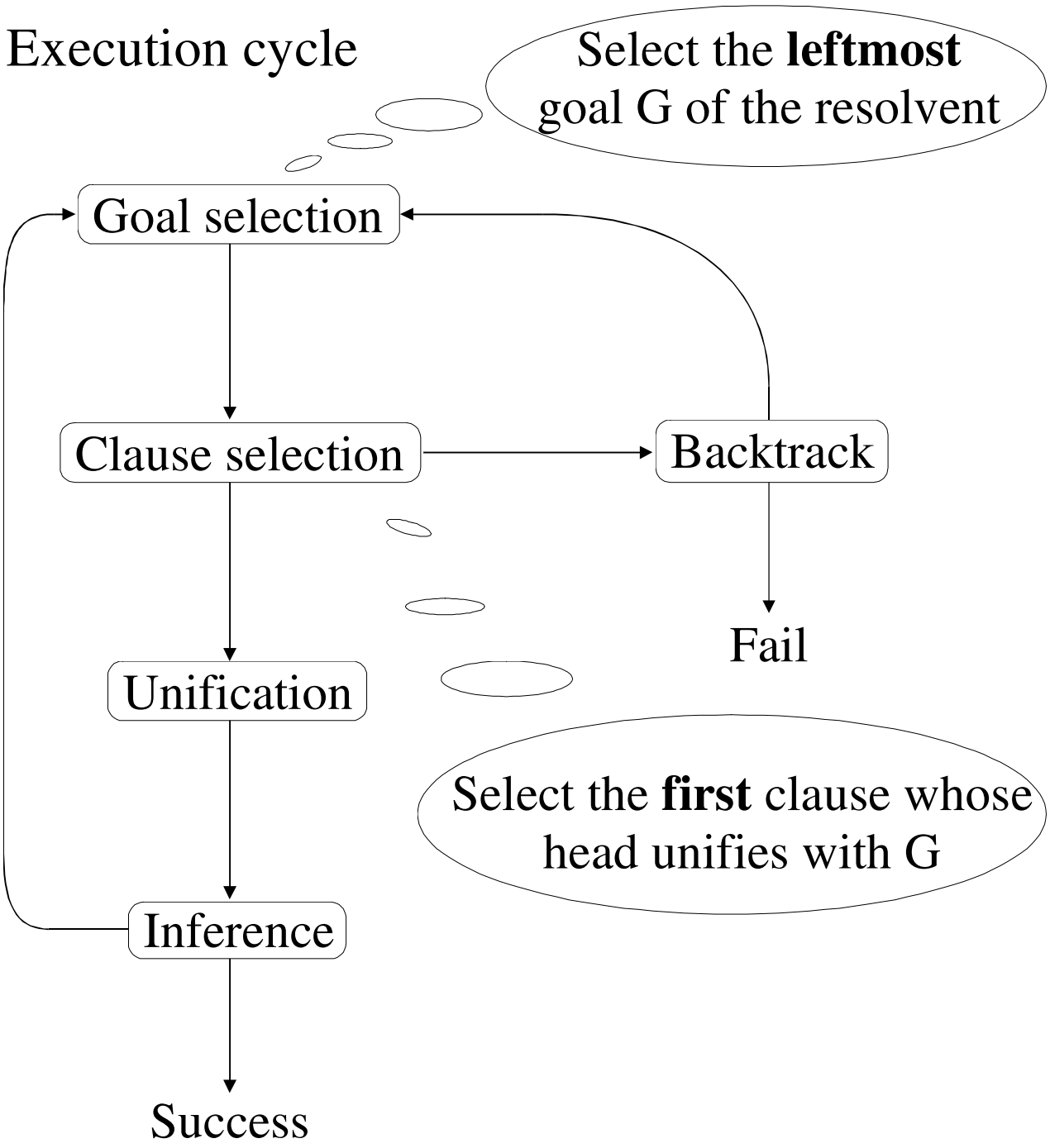
Heads, goals: atomic formulas

$p(t_1, t_2, ...t_i)$

Logic programming



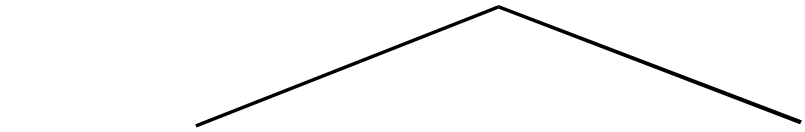
Prolog


Execution cycle
Select the **leftmost** goal G of the resolvent
Goal selection
Clause selection
Backtrack
Fail
Unification
Select the **first** clause whose head unifies with G
Inference
Success

The source of parallelism

any goal - leftmost goal
several goals in parallel = AND-parallelism

any clause - first clause
several clauses in parallel = OR-parallelism



OR-parallelism AND-parallelism

Dependent AND-parallelism

Independent AND-parallelism

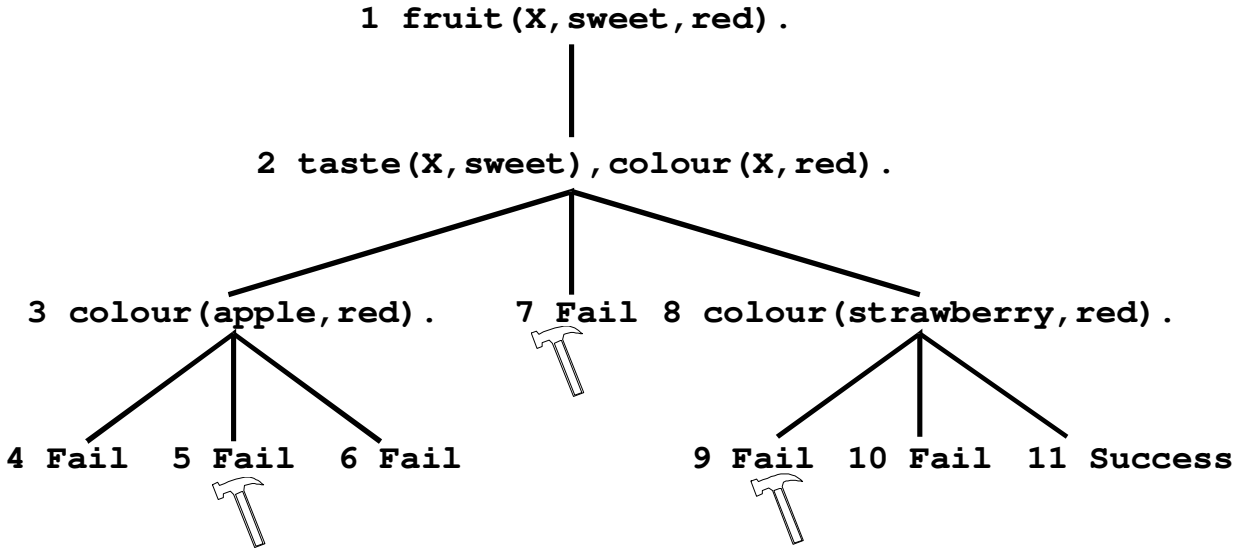
Pipeline AND-parallelism

Search parallelism

Path parallelism

OR-parallel logic computational models

:- fruit (X, sweet, red).
fruit (X, Y, Z) :- taste (X, Y), colour (X, Z).
taste (apple, sweet).
taste (lemon, acid).
taste (strawberry, sweet).
colour (lemon, yellow).
colour (apple, green).
colour (strawberry, red).



OR-parallel logic computational models

Problems and issues

Handling of multiple bindings

1. Efficient memory management
2. Efficient dereferencing of terms

Assigning work to workers

1. Controlling the granularity
2. Minimizing overhead at task switching
3. Handling side effects

Common techniques

Copying the stacks

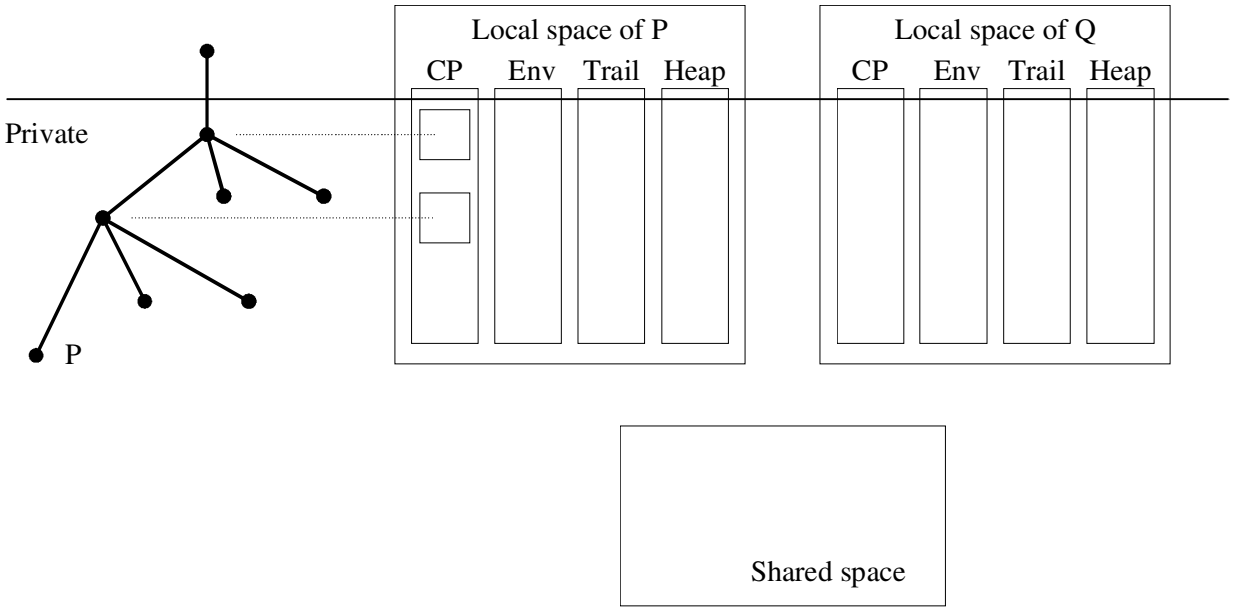
Sharing the stacks

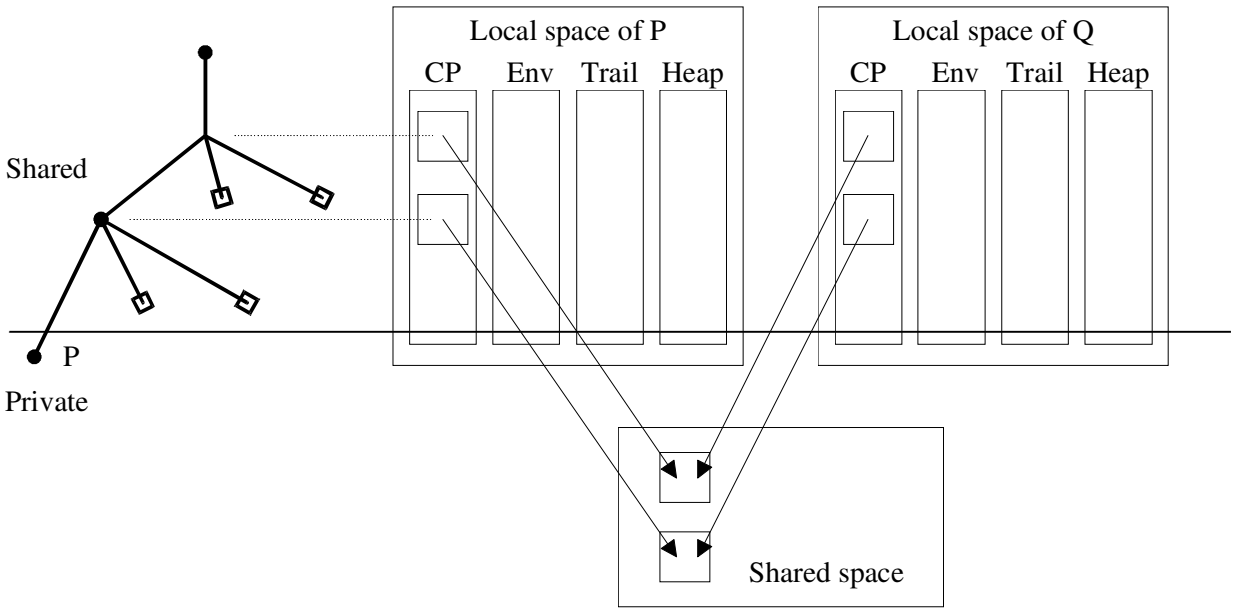
Recomputing the stacks

Closed binding environment

Copying the stacks

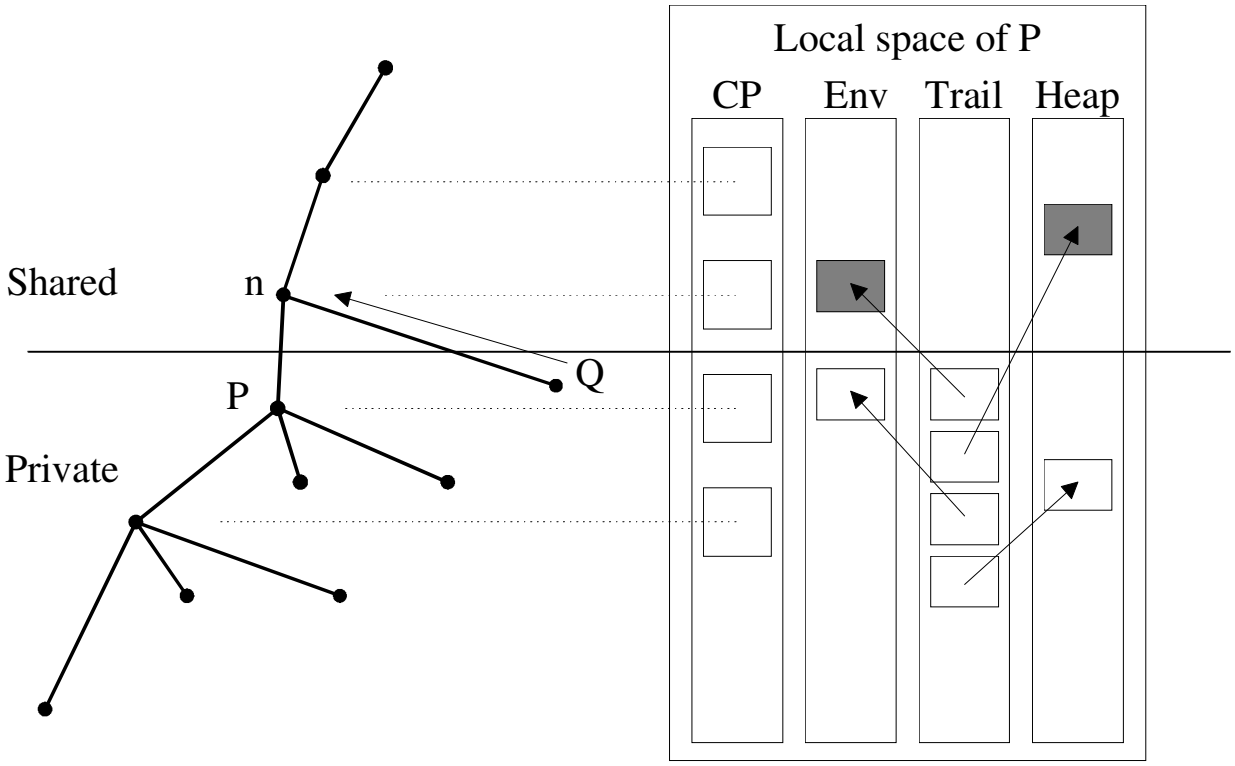
The basic execution model





Copying the stacks

Incremental copying

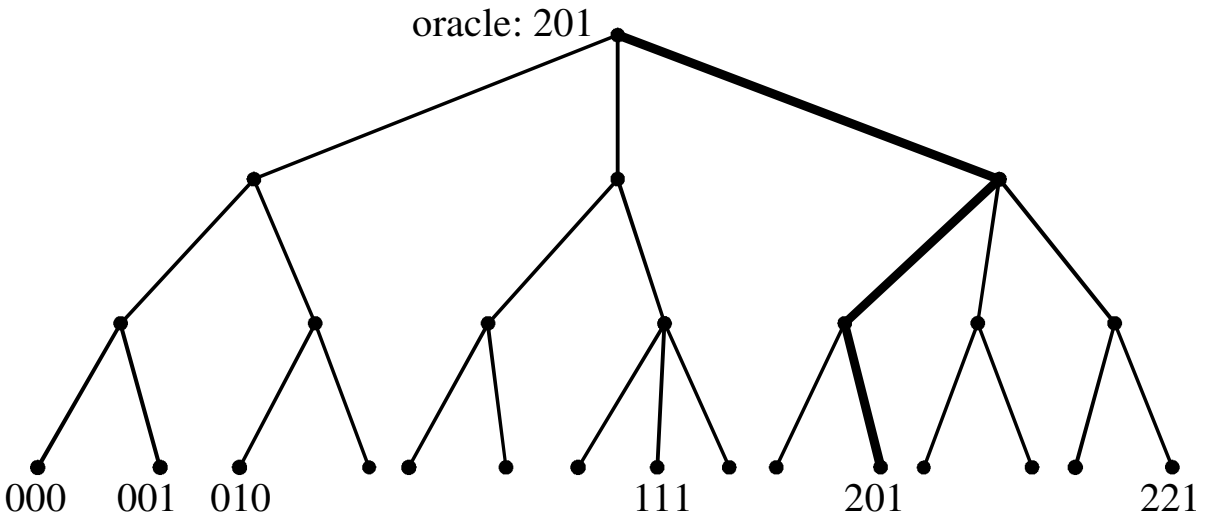


Recomputing the stacks

Motivation

On a heterogeneous network:

- Sharing is not always possible
- Large communication overhead at copying



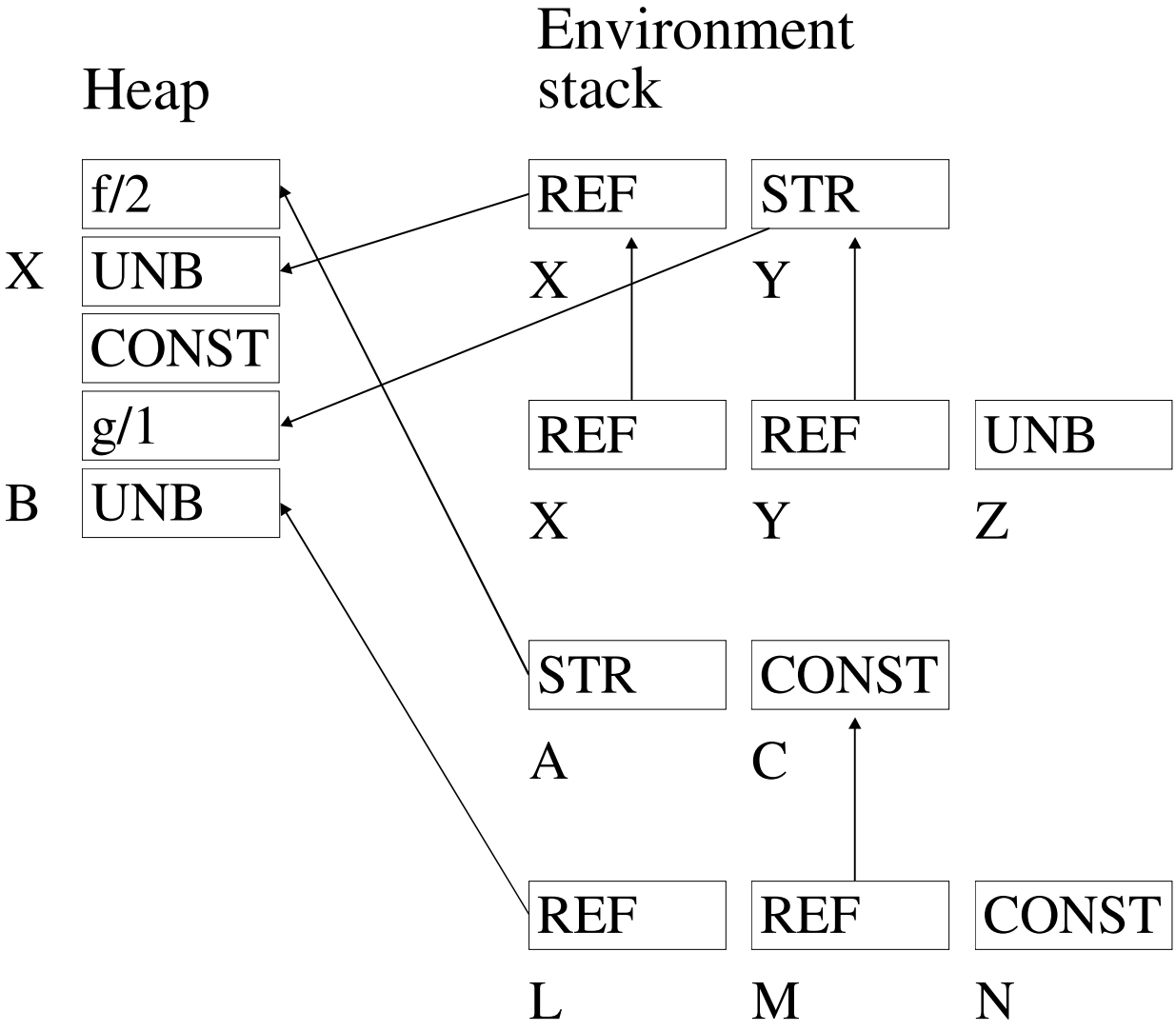
Closed binding environment

:- ..., p(X, Y),....

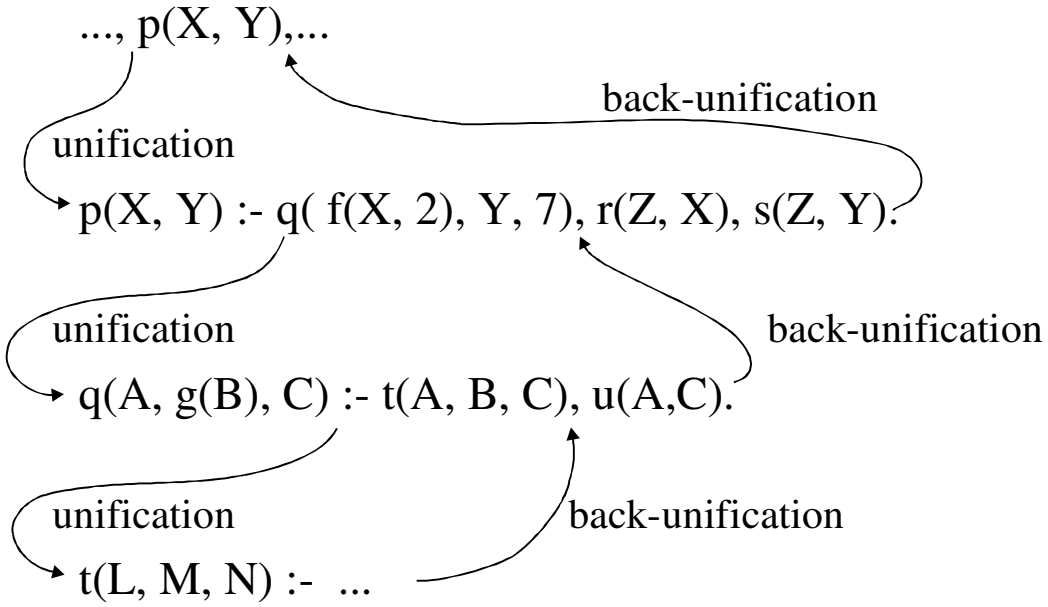
p(X, Y) :- q(f(X, 2), Y, 7), r(Z, X), s(Z, Y).

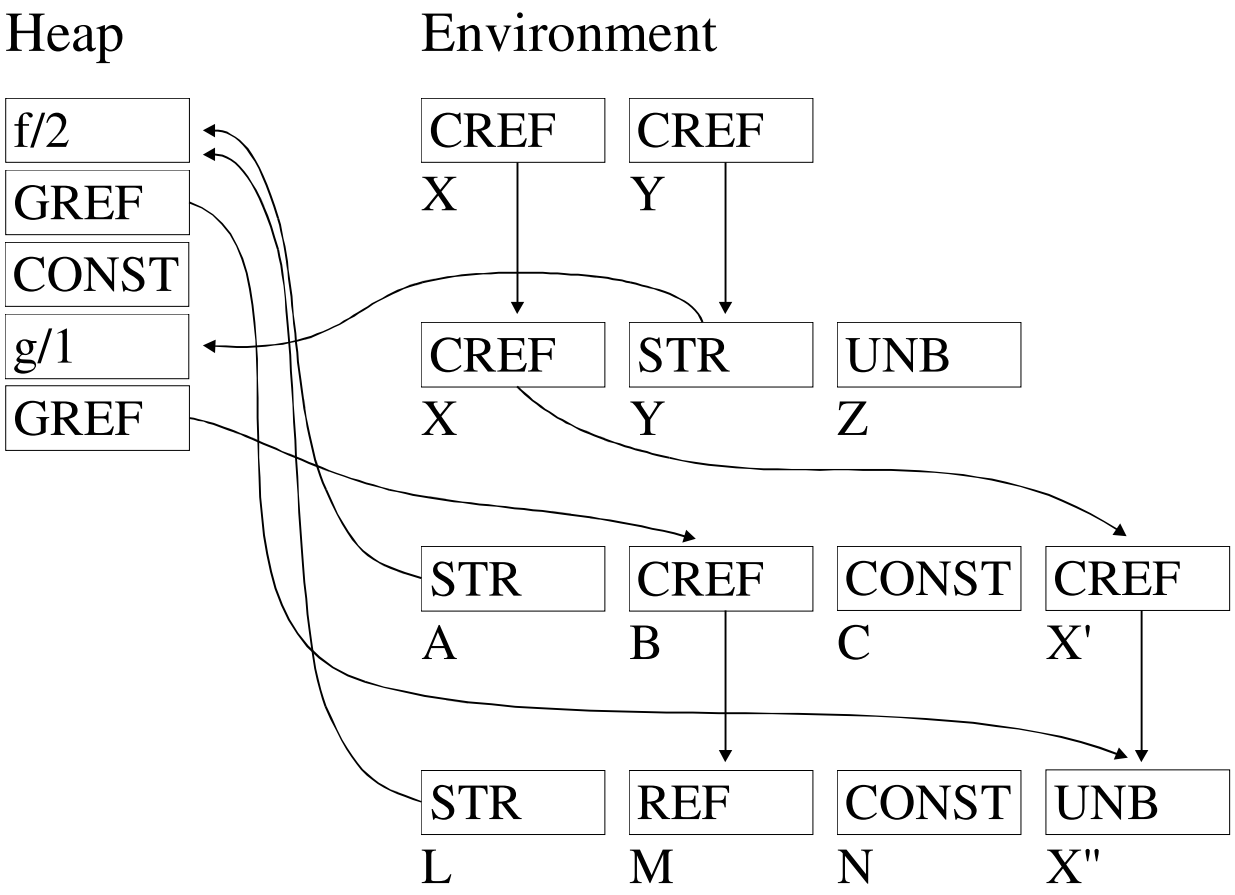
q(A, g(B), C) :- t(A, B, C), u(A, C).

t(L, M, N) :- ...

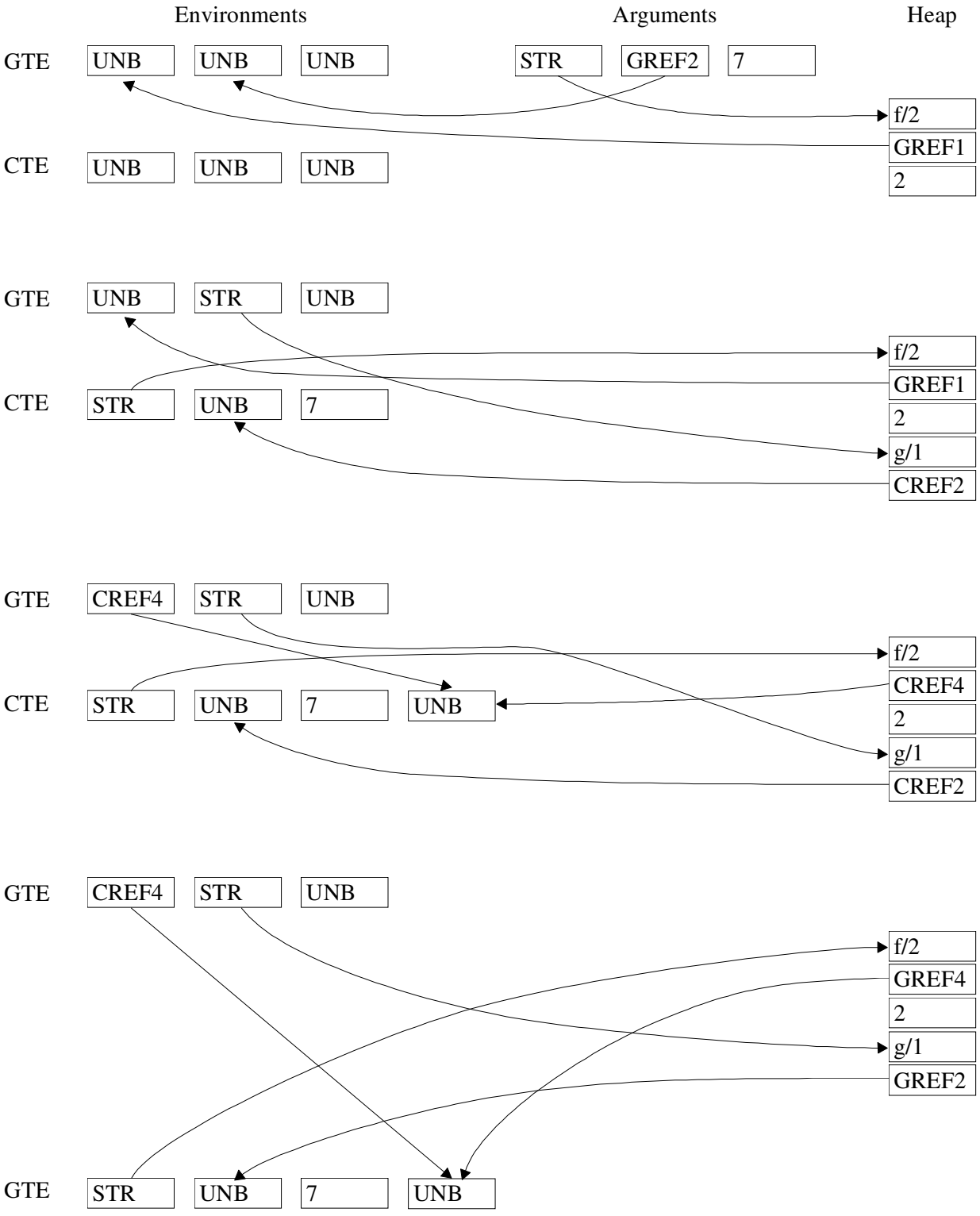


Closed binding environment

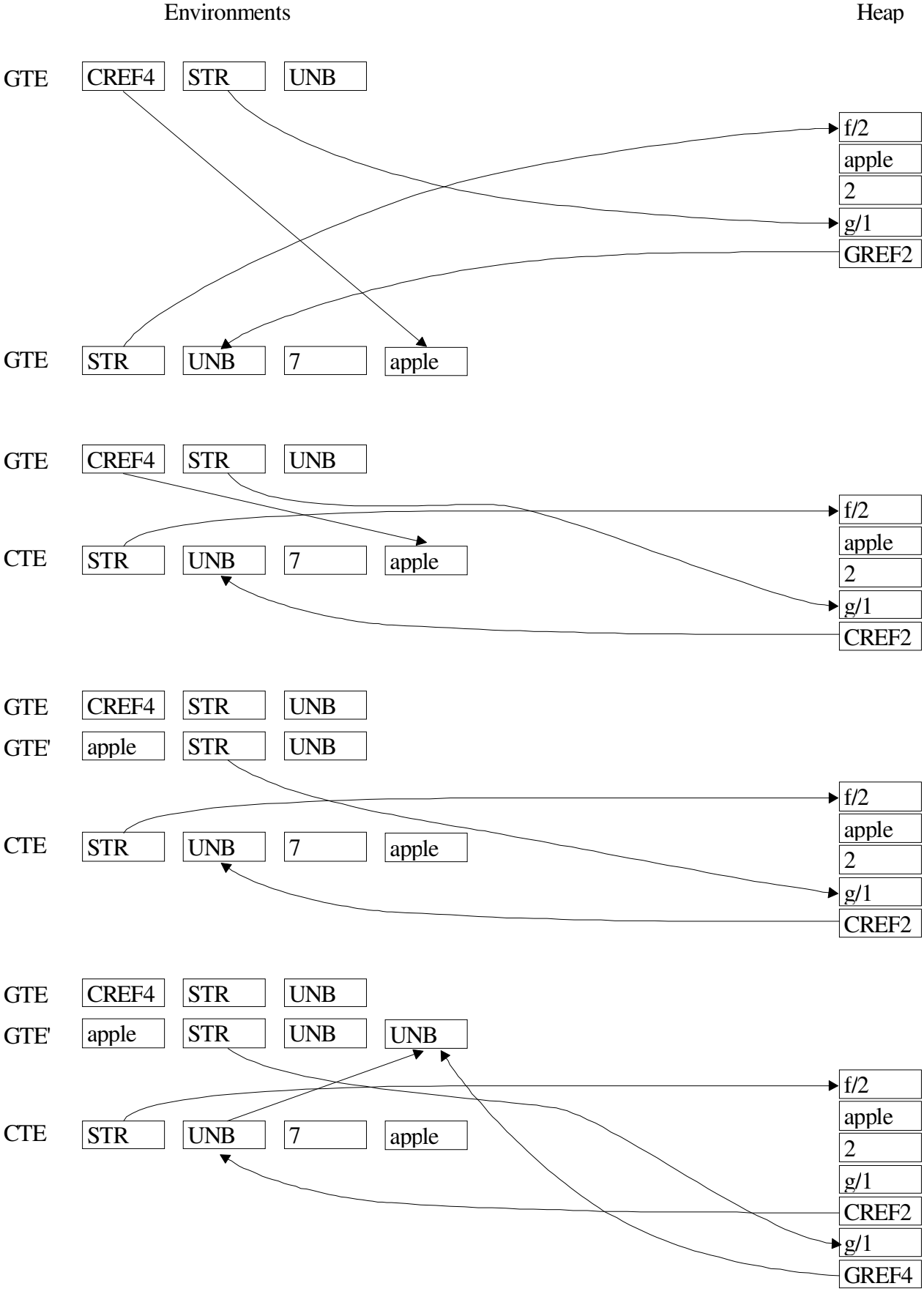




Closed binding environment



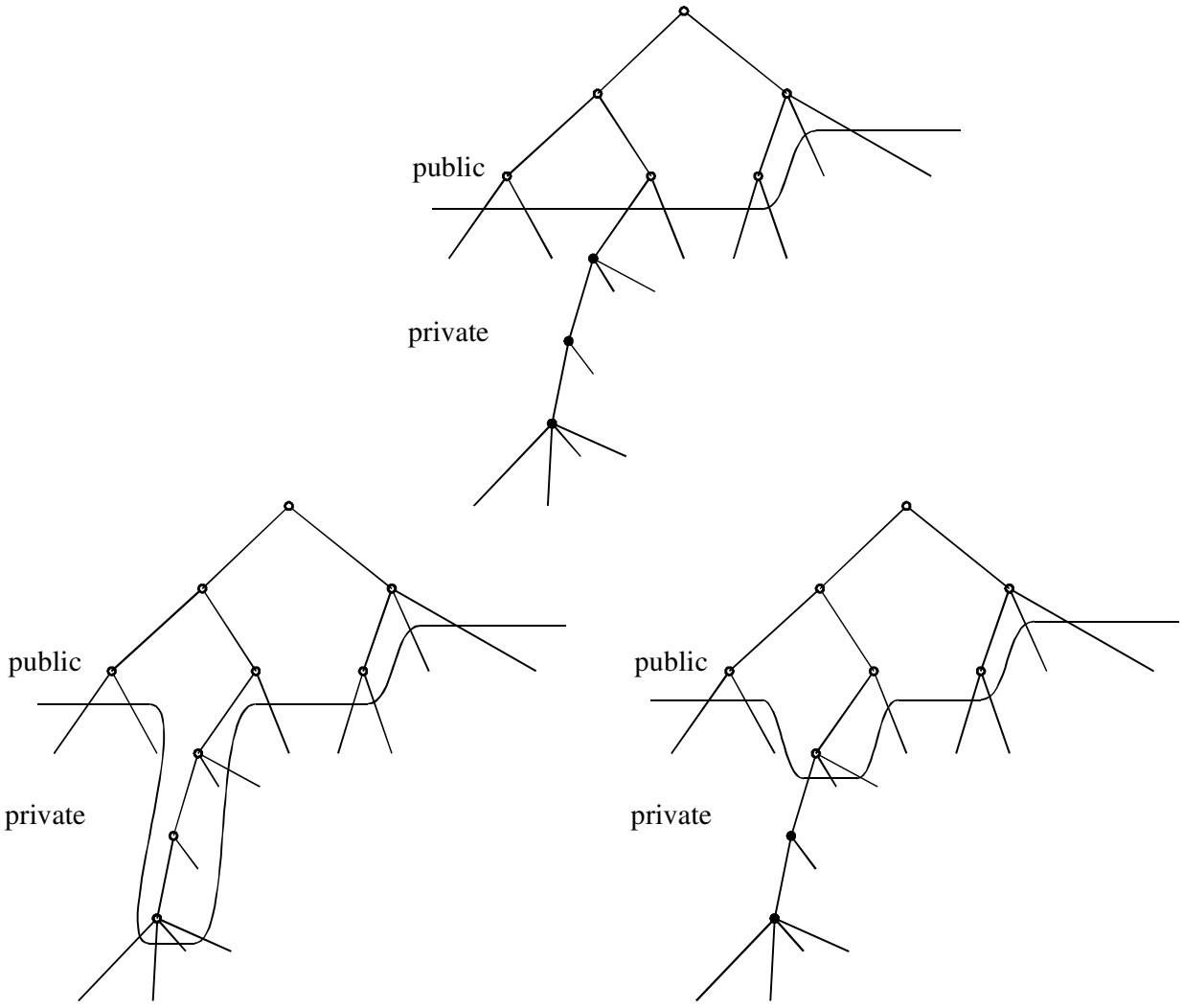
Closed binding environment



Scheduling

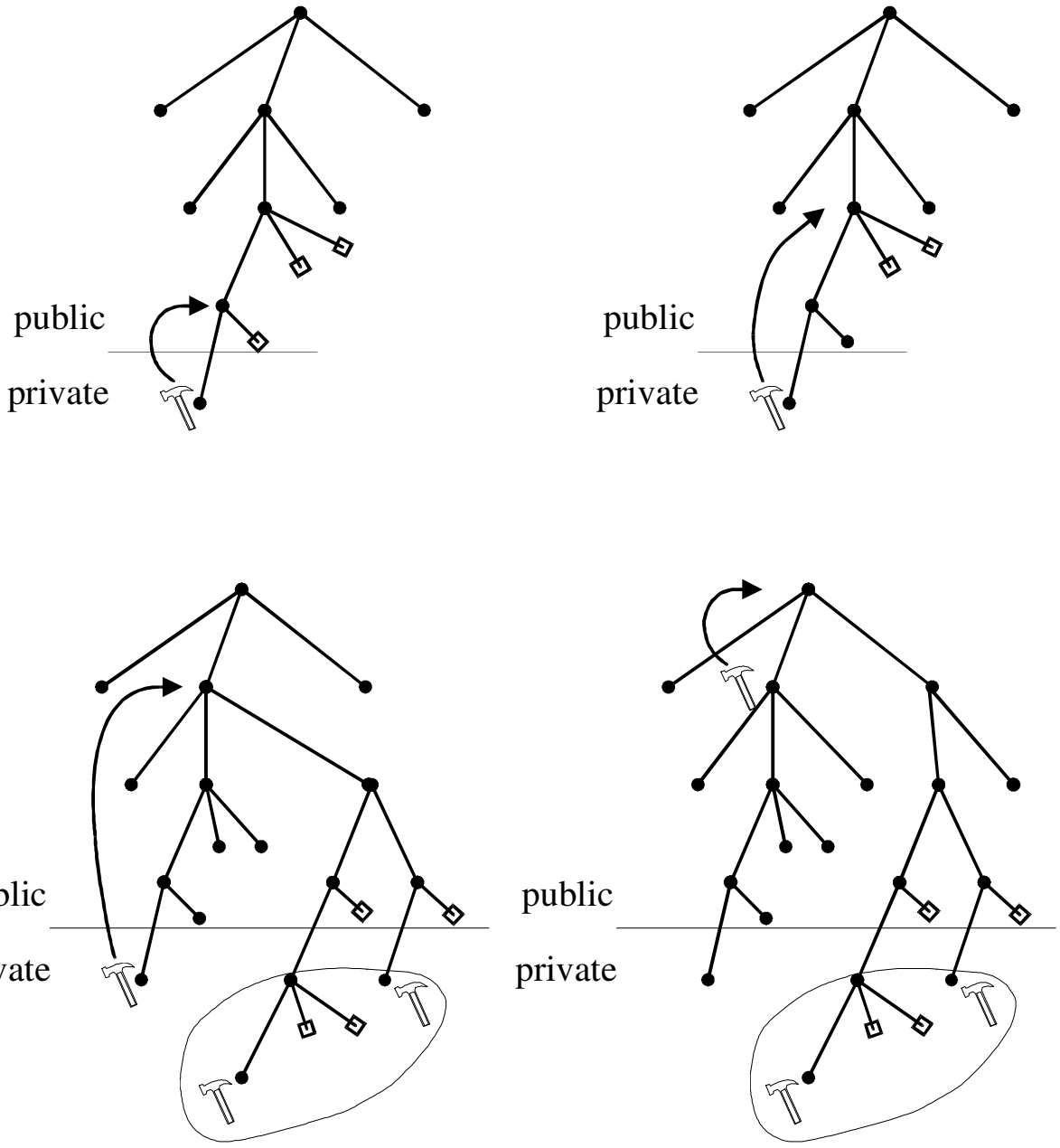
The aim of scheduling:

- providing workers with work
  - minimizing idle states
  - minimizing task switching cost
- controling granularity
- correct handling of side-effect predicates



Scheduling

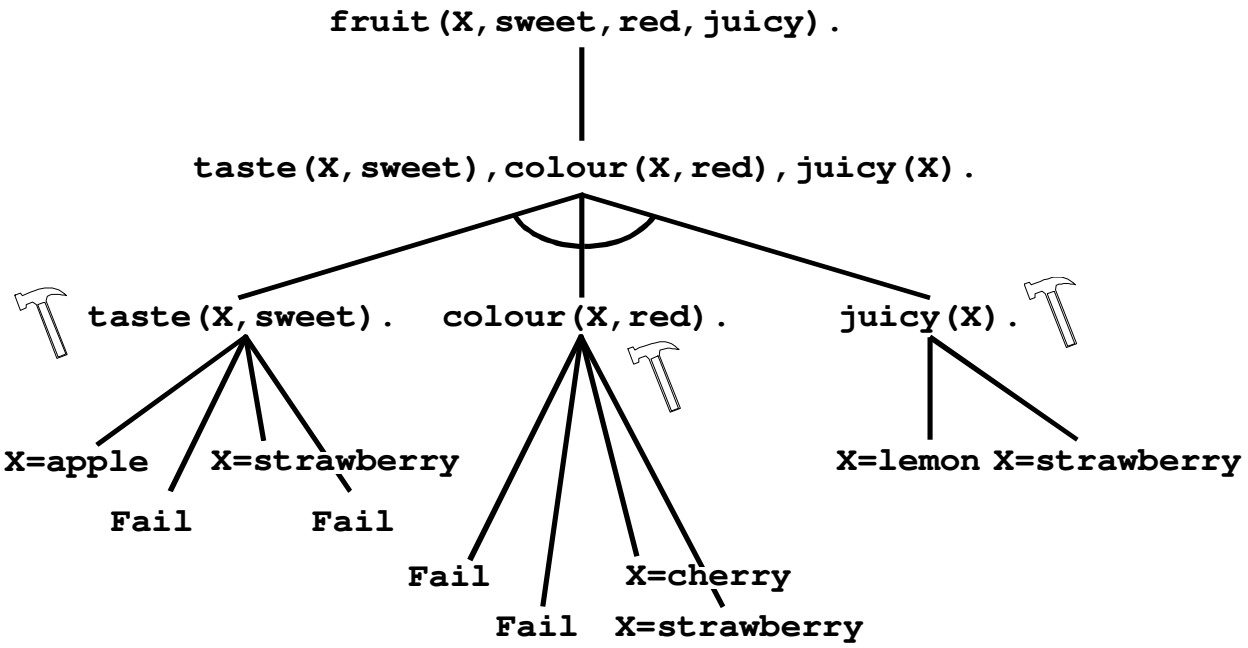
Muse: a bottom-most strategy



AND-parallel logic computational models

:- fruit (X, sweet, red, juicy).
fruit (X, Y, Z) :- taste (X, Y), colour (X, Z), juicy(X).

taste (apple, sweet).
taste (lemon, acid).
taste (strawberry, sweet).
taste (cherry, sour).
colour (lemon, yellow).
colour (apple, green).
colour (strawberry, red).
colour (cherry, red).
juicy (lemon).
juicy (strawberry).



AND-parallel logic computational models

Problems and issues

Workers may bind a variable to different values causing the binding conflict

Aim: obtaining coherent bindings for the variables shared by several goals

Common techniques

- Independent AND-parallelism
  - Explicit goal ordering
  - Implicit goal ordering
    - Dynamic dependency check
    - Static dependency check
    - Hybrid dependency check
- Dependent AND-parallelism

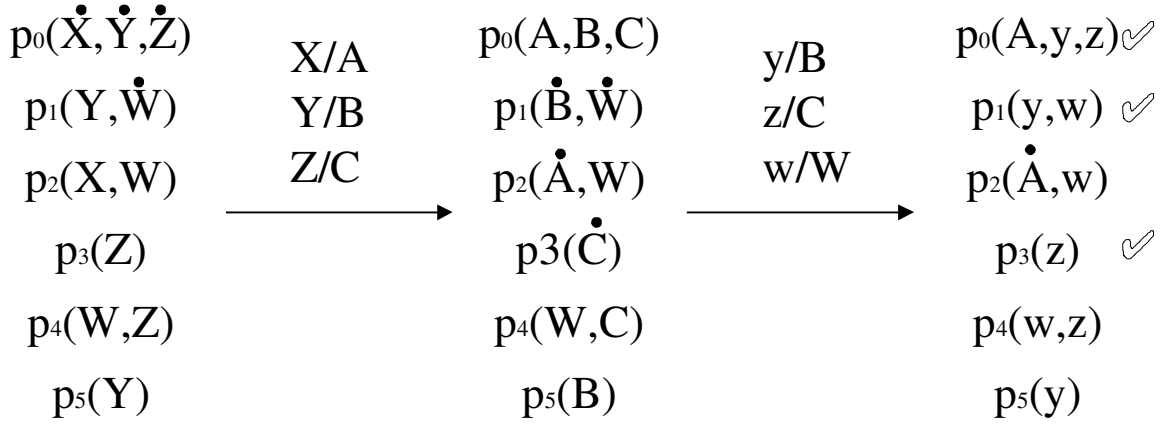
Independent AND-parallelism: dynamic approach

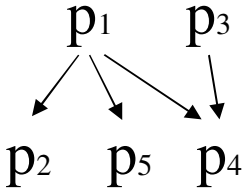
Forward execution: F-lists

:- $p_0(A,B,C)$.

$p_0(X,Y,Z)$:-$p_1(Y,W)$, $p_2(X,W)$, $p_3(Z)$, $p_4(W,Z)$, $p_5(Y)$.

F-list(W)=[p_2,p_4]
F-list(X)=[p_2]
F-list(Z)=[p_3,p_4]
F-list(Y)=[p_1,p_5]





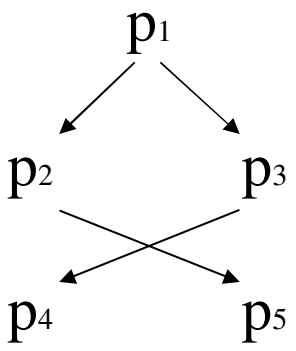
Independent AND-parallelism: dynamic approach

Backward execution: B-lists

$p_0(A,B,C)$:-$p_1(A)$, $p_2(A,B)$, $p_3(A,C)$, $p_4(C)$, $p_5(B,C)$.

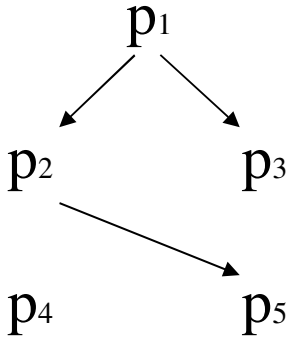
$p_1(a_1)$.
$p_2(a_1,b_1)$.
$p_2(a_1,b_2)$.
$p_3(a_1,c_1)$.
$p_3(a_1,c_2)$.
$p_4(c_1)$.
$p_5(b_1,c_2)$.
$p_5(b_2,c_1)$.

Before p_5 fails



p_1	SOLVED	$[p_0]$
p_2	SOLVED	$[p_1,p_0]$
p_3	SOLVED	$[p_1,p_0]$
p_4	SOLVED	$[p_3,p_0]$
p_5	EXECUTING	$[p_3,p_2,p_0]$

After backtracking

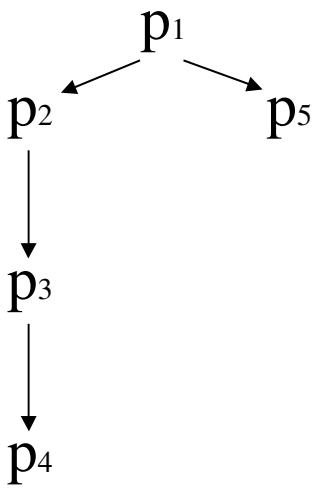


p_1	SOLVED	$[p_0]$
p_2	SOLVED	$[p_1,p_0]$
p_3	EXECUTING	$[p_2,p_1,p_0]$
p_4	GATHER	$[p_0]$
p_5	GATHER	$[p_2,p_0]$

Independent AND-parallelism: static approach

:- $p_0(A,B,C)$.
$p_0(X,Y,Z)$:-$p_1(Y,W)$, $p_2(X,W)$, $p_3(Z)$, $p_4(W,Z)$, $p_5(Y)$.

	Activation mode	Exit mode
p_0	(NGD,NGD,NGD)	
p_1	(NGI,NGI)	(G,NGI)
p_2	(NGI,NGI)	(NGI,G)
p_3	(NGI)	(G)
p_4	(G,G)	(G,G)
p_5	(G)	(G)



G: ground
NGD: non-ground dependent
NGI: non-ground independent

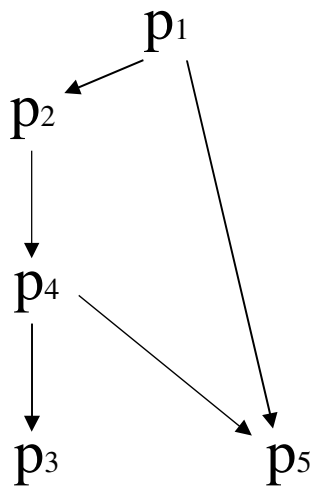
Independent AND-parallelism: hybrid approach

Forward execution

:- $p_0(V,V,z)$.

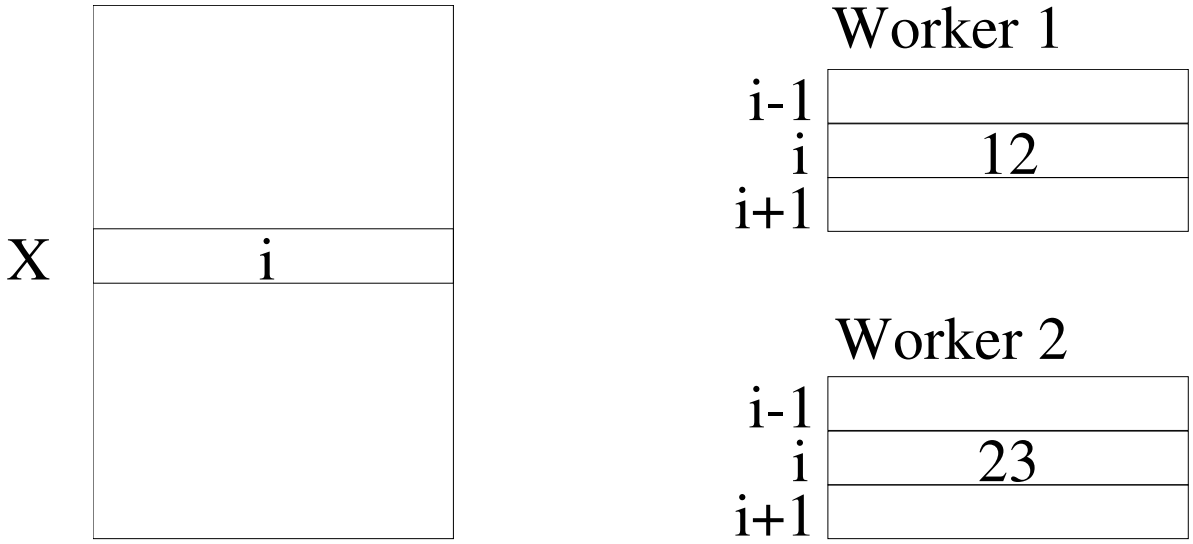
$p_0(X,Y,Z)$:-$p_1(Y,W)$, $p_2(X,W)$, $p_3(Z)$, $p_4(W,Z)$, $p_5(Y)$.

```
GPAR(Y,Z)
    SEQ   p1(Y,W)
          IF IPAR(Z,X)
               GPAR(W) p2(X,W) p4(W,Z)
               SEQ p2(X,W) p4(W,Z)
    IPAR(Z,Y) p3(Z) p5(Y)
```

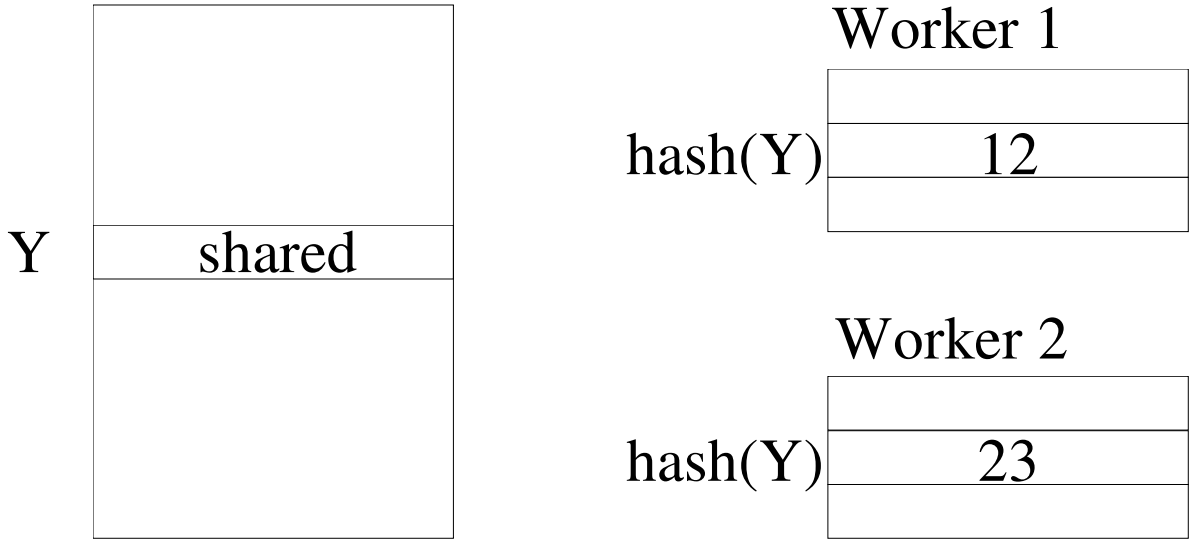


Sharing the stacks

Binding array

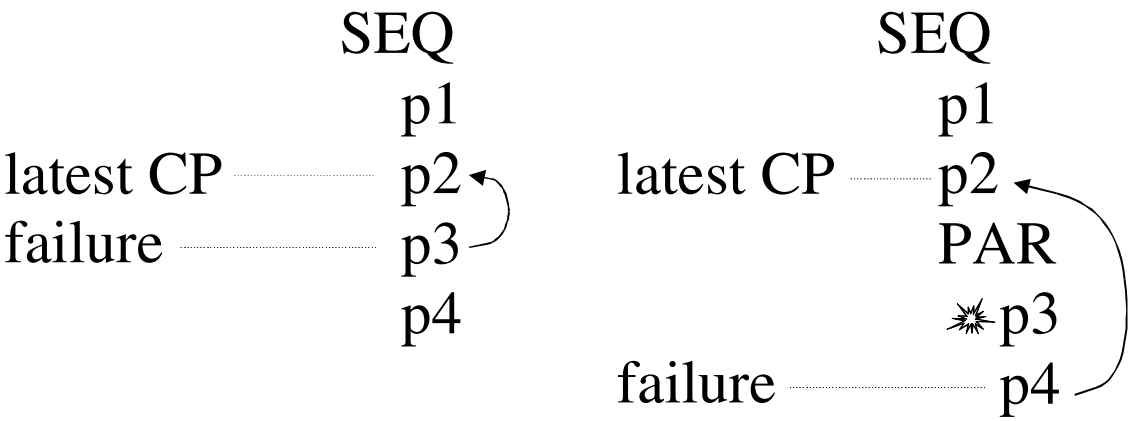


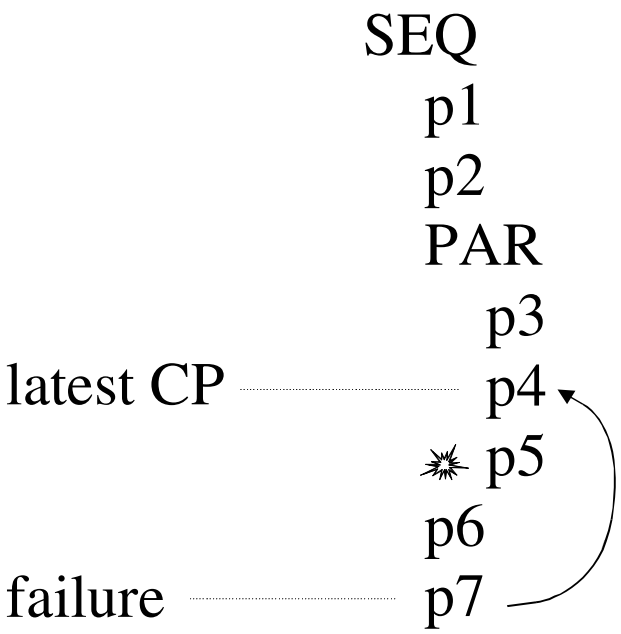
Hash windows



Independent AND-parallelism: hybrid approach

Backward execution





Introduction to λ-calculus

λ-expressions

Constants: 1,2, TRUE, FALSE, 'a', 'b', etc.
Variables: x, y, z
λ-expressions: F E
Primitive functions: + - AND, OR, etc.
λ-abstraction: λ<variable>.<λ-expression>

(λx.λy. + x ((λx. - x 3) y)) 5 6

Bound and free variables

An occurrence of a variable is bound if there is an enclosing λ-abstraction which binds it

λx.λy. + x ((λx. - x 3) y) + **x** ((λx. - x 3) **y**)

λy. + **x** ((λx. - x 3) y) (λx. - x 3) **y**

Introduction to λ-calculus

Reduction

executing a functional program = evaluating an expression
evaluation = selecting a reducible expression + reducing it
reducible expression (redex) = all arguments are present

+ (* 3 4) (- 8 3) not a redex

→ + 12 (- 8 3)

→ + 12 5

→ 7

β-reduction

applying a λ-abstraction to an argument

(λx.λy + x ((λx. - x 3) y)) 5 6

→β λy. + 5 ((λx. - x 3) y) 6

→β + 5 ((λx. - x 3) 6)

→β + 5 (- 6 3)

→ + 5 3

→ 8

Introduction to λ-calculus

β-abstraction

β-reduction can be applied backwards

$+\ 12\ 5 \leftarrow (\lambda x.\ +\ x\ 5)\ 12$

Conversion

name changing: α-conversion

$(\lambda x.\ +\ x\ 5) \leftrightarrow_{\alpha} (\lambda y.\ +\ y\ 5)$

β-reduction and β-abstraction: β-conversion

$\lambda y.\ +\ 5\ ((\lambda x.\ -\ x\ 3)\ y)\ 6 \leftrightarrow_{\beta} +\ 5\ ((\lambda x.\ -\ x\ 3)\ 6)$

applying built-in functions: δ-conversion

$+\ 7\ 5 \leftrightarrow_{\delta} +\ (-\ 12\ 5)\ 5$

two literally different expressions 'ought' to be equal: η-conversion

if x is not free in F and F is a function:

$\lambda x.\ (F\ x) \leftrightarrow_{\eta} F$

$\lambda x.\ +\ 1\ x \leftrightarrow_{\eta} +\ 1$

Introduction to λ-calculus

Normal forms

if there are no more redexes, the expression is said to be in normal form

not every expression has a normal form (⊥)
$(\lambda x.\ x\ x)(\lambda x.\ x\ x) \rightarrow (\lambda x.\ x\ x)(\lambda x.\ x\ x) \rightarrow \ldots$

Weak Head Normal Form (WHNF)

a λ-expression containing no top-level redexes

$F\ E_1\ E_2\ \ldots\ E_n$ $(n \geq 0)$ F is either a variable or data object or F is a λ-abstraction or primitive function and $F\ E_1\ E_2\ \ldots\ E_m$ is not a redex for any $m \leq n$

Head Normal Form (HNF)

a λ-abstraction of form $\lambda x_1.\lambda x_2 \ldots \lambda x_n.\ M_1\ M_2\ \ldots M_m$ $(n,m \geq 0)$ and M_i is a variable or data object or $M_1\ M_2\ \ldots\ M_p$ is not a redex for any $p \leq m$

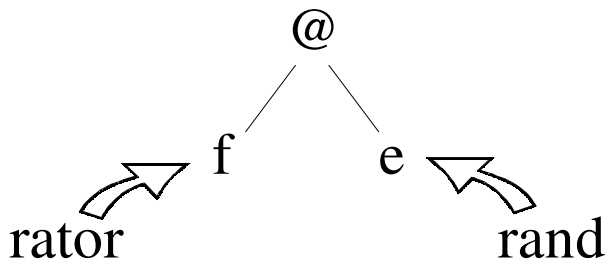
Introduction to λ-calculus

Strictness

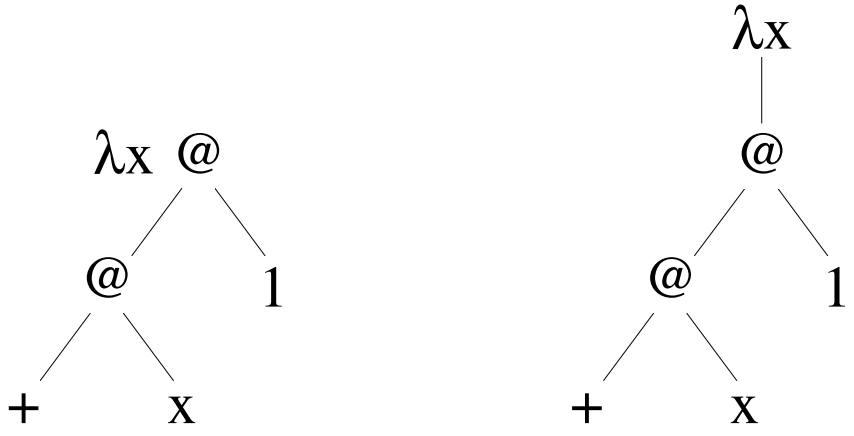
a function f is strict if and only if
f ⊥ = ⊥

Representing the expressions: abstract trees

application: f e



λ-abstraction: λx. + x 1



Variables, numbers, built-ins are always leaves.

Introduction to λ-calculus

Implementation issues

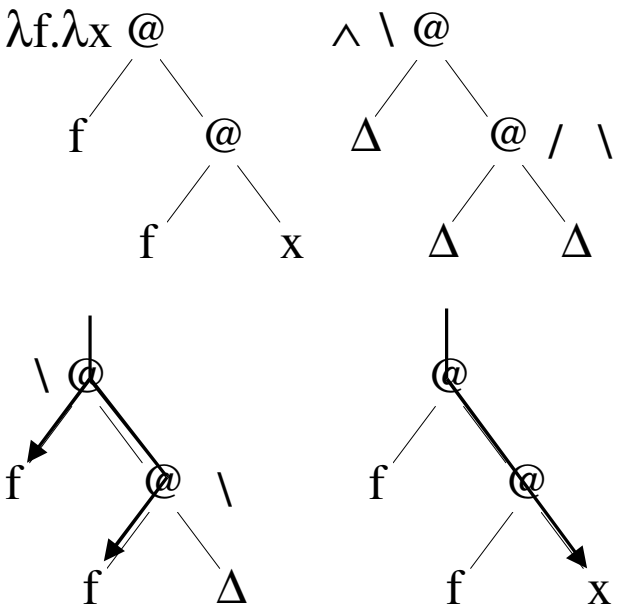
Combinators

a λ-expression which contains no free variables

Supercombinators

S$: a λ-expression of form $\lambda x_1.\lambda x_2...\lambda x_n$. E and E is not a λ-abstraction, $S does not contain free variables and λ-abstractions in E are supercombinators

Director string terms



Director strings:
- ∧ both rator and rand
- / only the rator
- \ only the rand
- \- neither

contains bound variable

Δ is a placeholder.

Reduction strategy

A redex is only potentially reducible

The selected redex must be a strategically reducible expression (s-redex)

Strategically normal expression (s-normal) contains no s-redexes

Strategy in CTDNet2: Innermost Context-Free Applicative Reduction (ICFAR)

a redex is ICFA-reducible if and only if it is context-free and both rator and rand are ICFA-normal

an expression is ICFA-normal which is s-normal under ICFAR

normality $\rightarrow$ ICFA-normality

ICFA-reducible examples:
$(\lambda y.y\ 2)$ $(\lambda x.\ x\ \lambda x.(\lambda y.\ y\ x))$ $(\lambda x.\ (\lambda y.\ (+\ y\ x)\ x)\ 4)$

Evaluation strategies

Eager and lazy evaluation

eager evaluation:

call by value

evaluate arguments before call to procedure

lazy evaluation:

call by need

evaluate arguments when really needed

Eager *Lazy*

most constructs are strict **conditionals**

if E then F else G

two expressions should

be evaluated at most

I/O modelled as streams

infinite lists are

never fully instantiated

Evaluation strategies

Selecting the next redex

If there are more than one redexes,
any of them can be chosen

Church-Rosser Theorem I:
at most one normal form exists for an expression

Church-Rosser Theorem II:
if normal form exists, the reduction
can be achieved with normal order

Normal order		*Applicative order*
leftmost, outermost redex reduced first		rand is reduced to normal form before being passed to rator
call-by-need		call-by-value
reduction results in normal form (if exists)		reduction may never terminate
nothing is evaluated unnecessarily		unnecessary work may be done
purely sequential		offers parallelism

Argument Handling

String reduction

copies of rand are substituted into rator

each copy is evaluated separately

Graph reduction

copies of a pointer to the rand are substituted into rator

rand is evaluated once

String reduction	*Graph reduction*
copies of arguments are a waste of space	need to copy rator (less space needed)
a redex is evaluated as many times as many copies exist	a redex is evaluated once
	garbage collection is essential
eager scheme	lazy scheme

Parallel graph reduction

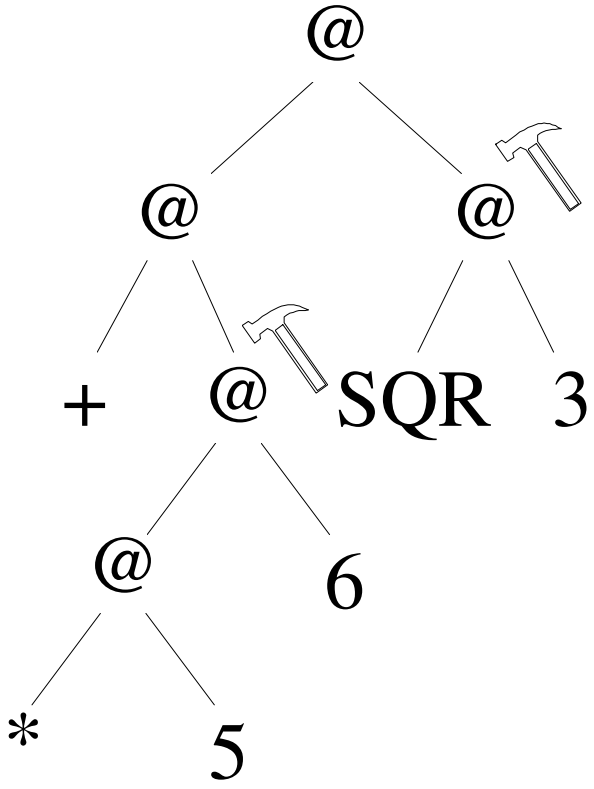
Graph reduction is an inherently parallel activity

Graph reduction is an inherently distributed activity

Communication is mediated through the graph

The state of computation is well defined
- the state of the graph

- type of parallelism
- task control
- granularity control
- synchronisation



Parallel graph reduction

Conservative parallelism

a new task is created only if its result is really needed

+ x y x and y evaluated in parallel

if c then x else y x and y not evaluated in parallel

Speculative parallelism

all possible tasks are activated

if c then x else y c,x and y evaluated in parallel

Conservative parallelism

no work is done in vain

fails to exploit the available parallelism

strictness analysis

Speculative parallelism

exploits the parallelism

large amount of resources are allocated for tasks which will be discarded later

Parallel graph reduction

Granularity

too small tasks: spawning a task is a costly operation
task control overhead may dominate

too large tasks: too little concurrency,
work is distributed among agents unevenly

Synchronisation

recall: laziness - everything is evaluated once

two tasks may try to evaluate the same piece of graph

Blocking mechanism

the first task marks the traversed nodes

after operation the graph is replaced by the result and the second task may proceed without repeating the computation

optimization: an expression in WHNF will not be changed, therefore multiple tasks may have access to it

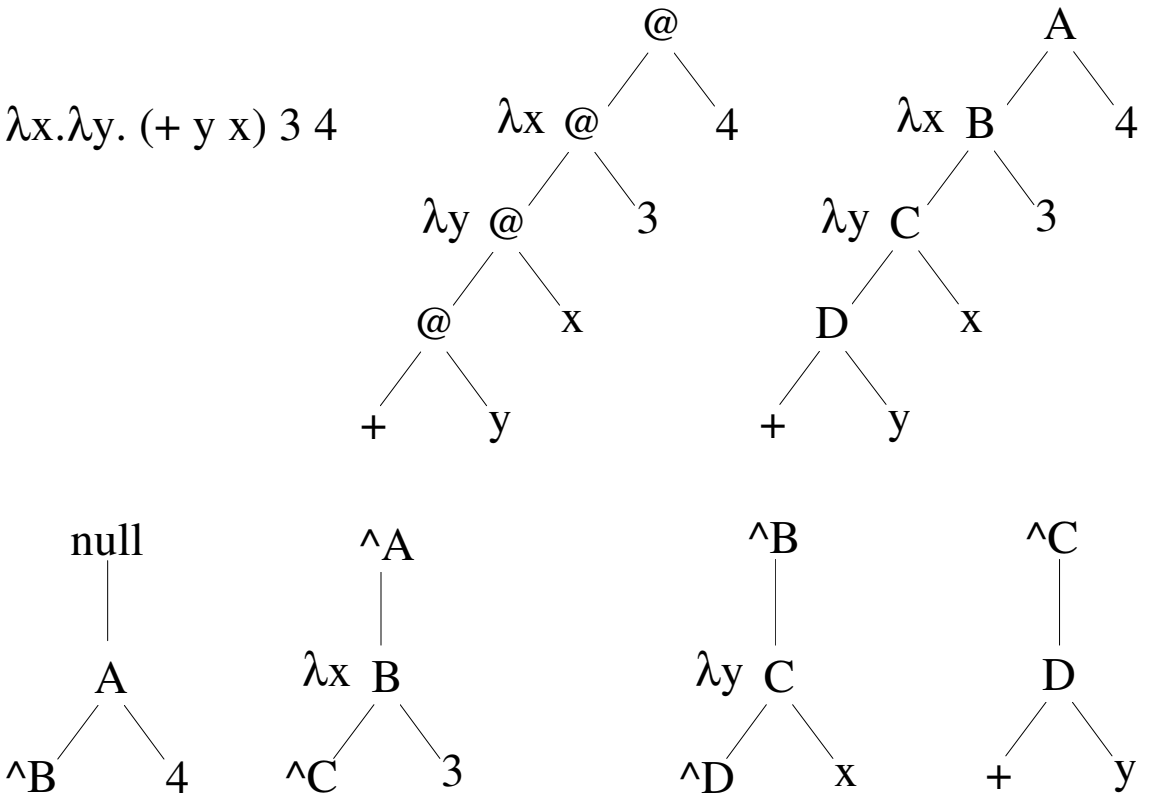
A case study: CTDNet2

CTDNet2:

- an evaluation framework based on applicative reduction
- lazy reduction subsystems: conditionals, recursion, lists
- process based implementation
- director string notation
- garbage collection

Process model in CTDNet2

λx.λy. (+ y x) 3 4

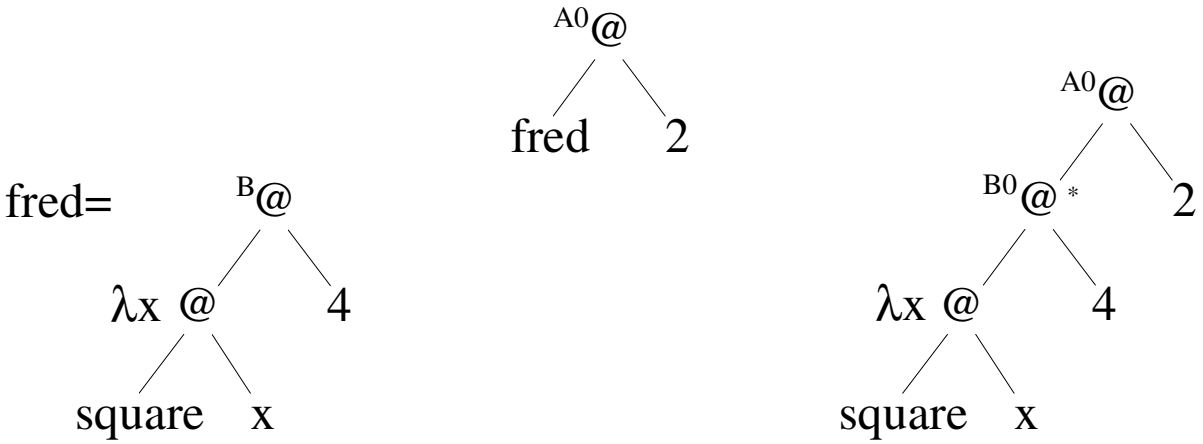


Reduction cycle

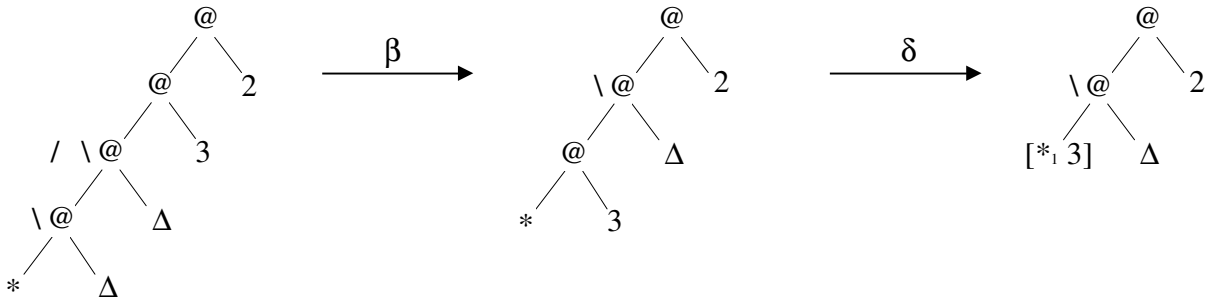
Passive rator

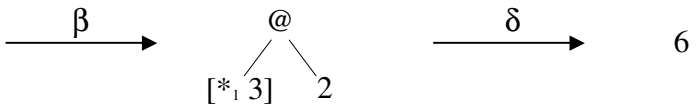
Definitions: passive processes

Passive processes must be activated before reduction



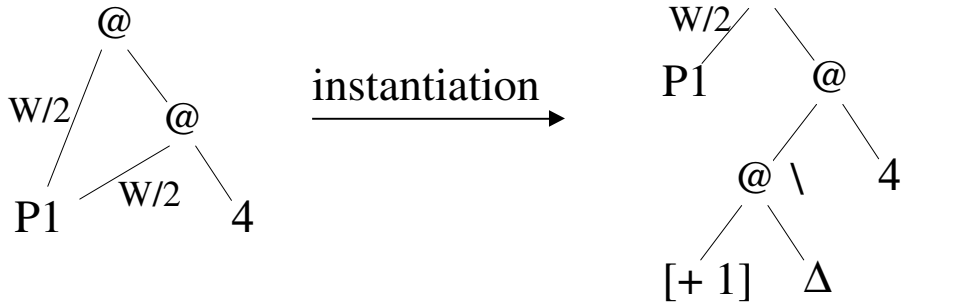
Unshared active rator





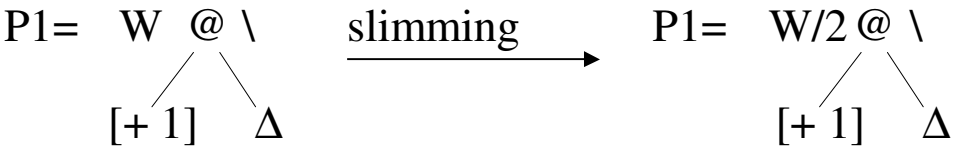
Reduction cycle

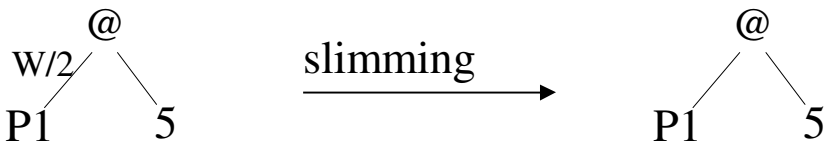
Shared active rator



P1 has weight W

P1 has weight W/2



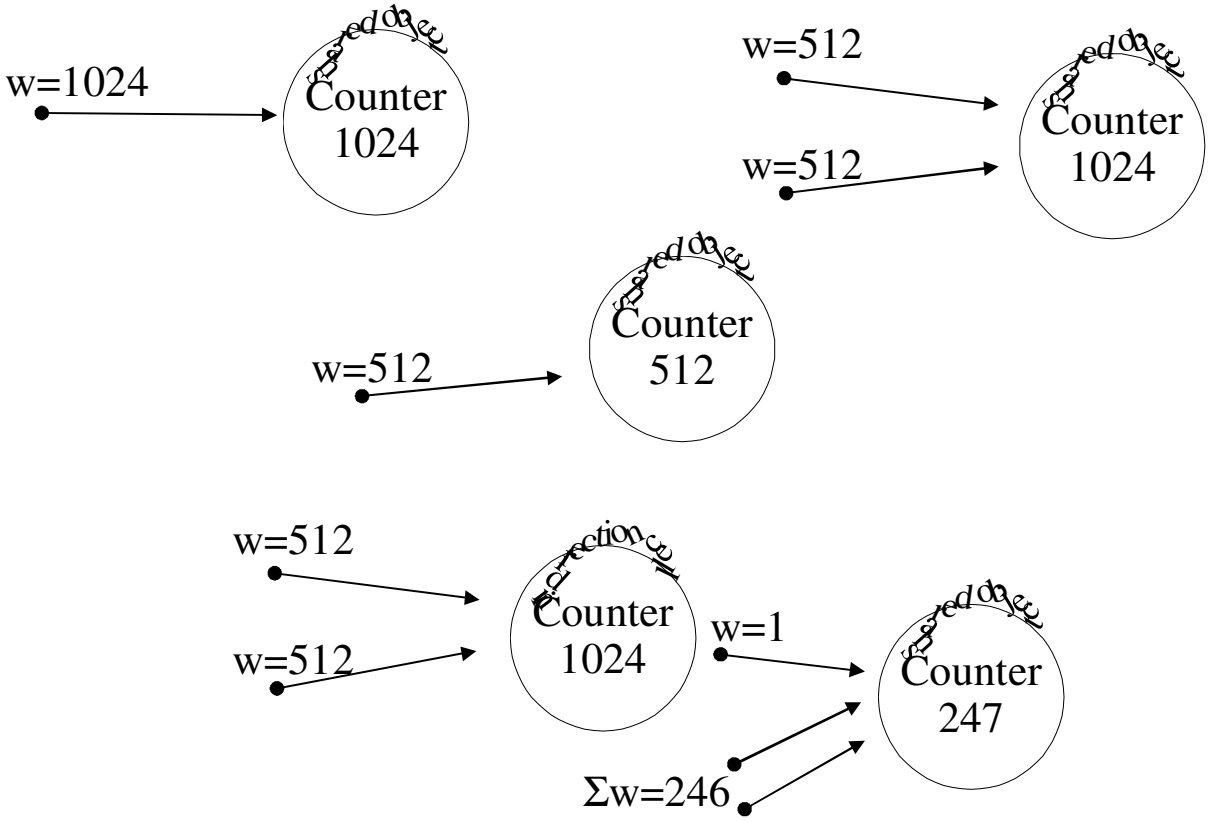


P1 has weight W/2

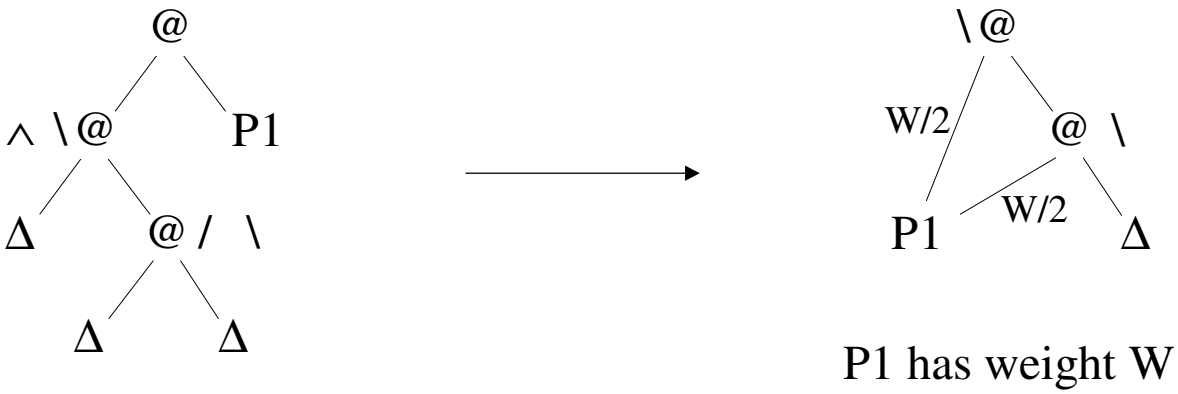
P1 has no weight

Reduction cycle

Garbage collection



Unshared active rand



P1 has weight W