Parallel Logic Programming Languages

Ian Foster &
Steve Gregory
Imperial College
London

Contents

Parallel Logic Languages

Horn clause logic:

$H \leftarrow B_1, B_2, \ldots, B_n.$

Declarative interpretation:

H is true if all B_i are true.

Procedural interpretation:

to prove H, prove all B_i

Process interpretation:

H reduces to a network of concurrent processes B_i.

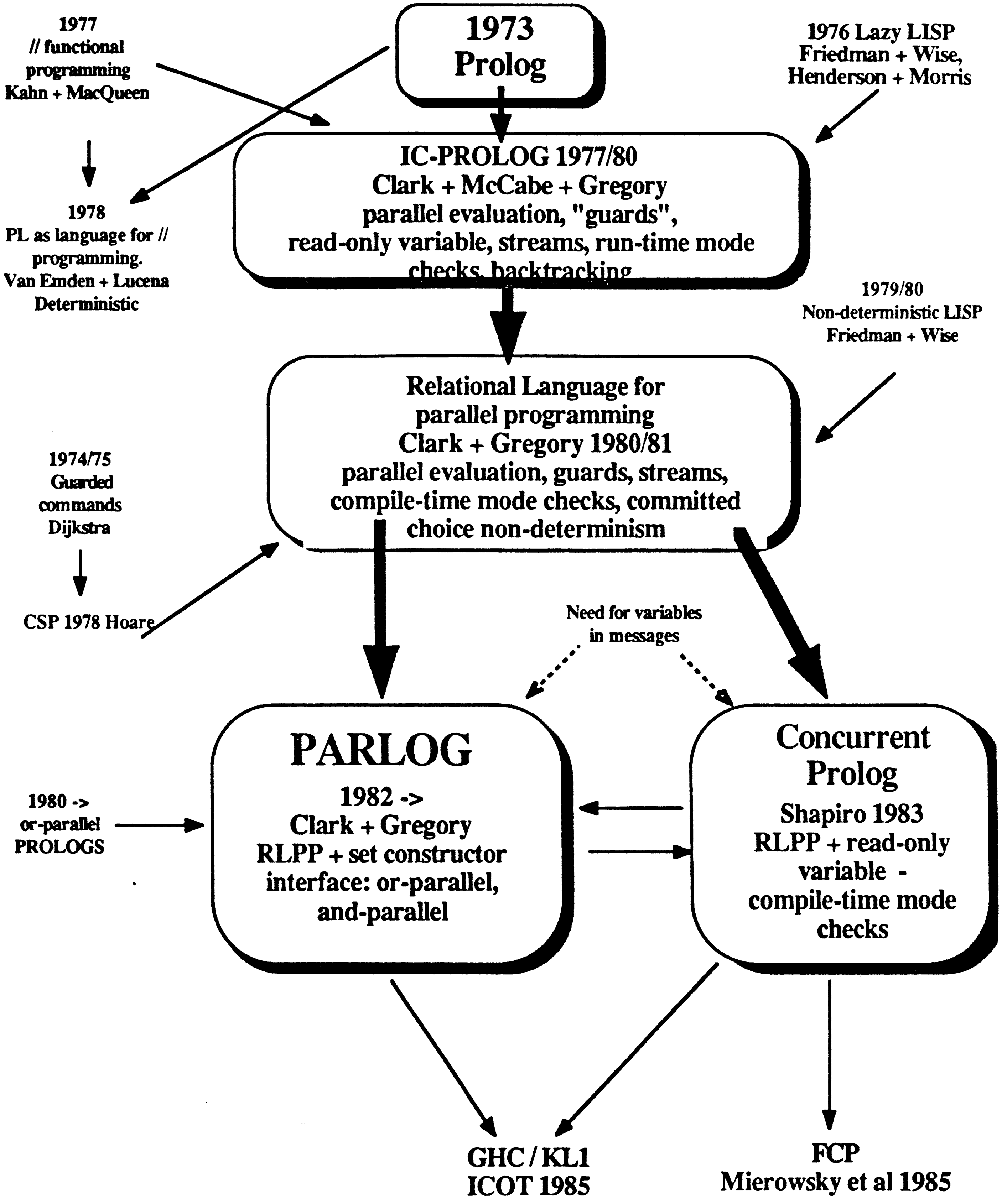
Parallel logic languages =

Horn clause logic + AND-parallelism
communication
synchronization
committed-choice non-determinism

They are interesting because they are:

logic programming languages
parallel languages
expressive

Parallel Logic Programming
A Partial History



1973
Prolog

IC-PROLOG 1977/80
Clark + McCabe + Gregory
parallel evaluation, "guards",
read-only variable, streams, run-time mode checks, backtracking

Relational Language for parallel programming
Clark + Gregory 1980/81
parallel evaluation, guards, streams, compile-time mode checks, committed choice non-determinism

PARLOG
1982 ->
Clark + Gregory
RLPP + set constructor interface: or-parallel, and-parallel

Concurrent Prolog
Shapiro 1983
RLPP + read-only variable - compile-time mode checks

GHC / KL1
ICOT 1985

FCP
Mierowsky et al 1985

1977
// functional programming
Kahn + MacQueen

1978
PL as language for // programming.
Van Emden + Lucena
Deterministic

1974/75
Guarded commands
Dijkstra

CSP 1978 Hoare

1980 ->
or-parallel PROLOGS

1976 Lazy LISP
Friedman + Wise,
Henderson + Morris

1979/80
Non-deterministic LISP
Friedman + Wise

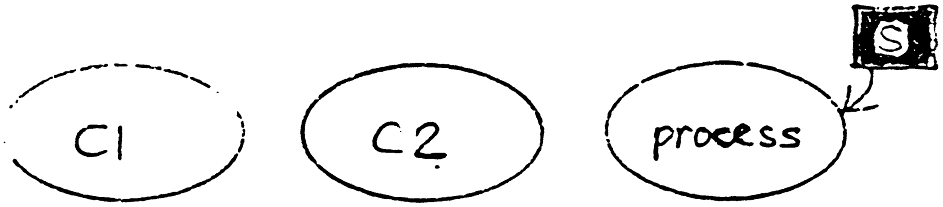
Need for variables in messages

AND-PARALLELISM

Relation call = Process

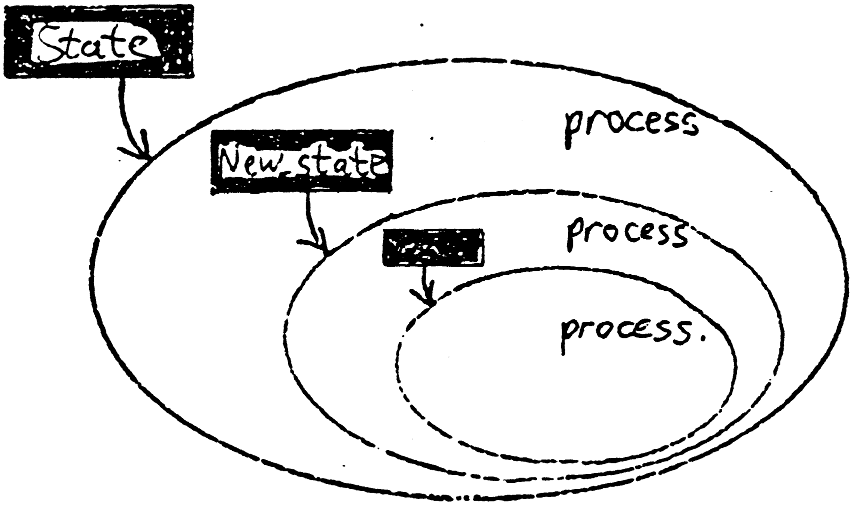
Conjunction = Process network

e.g. C1, C2, process(s)



Clause with one call in body =
Change of process state

```
mode process(?).
e.g. process(State) ← process(New_state).
```

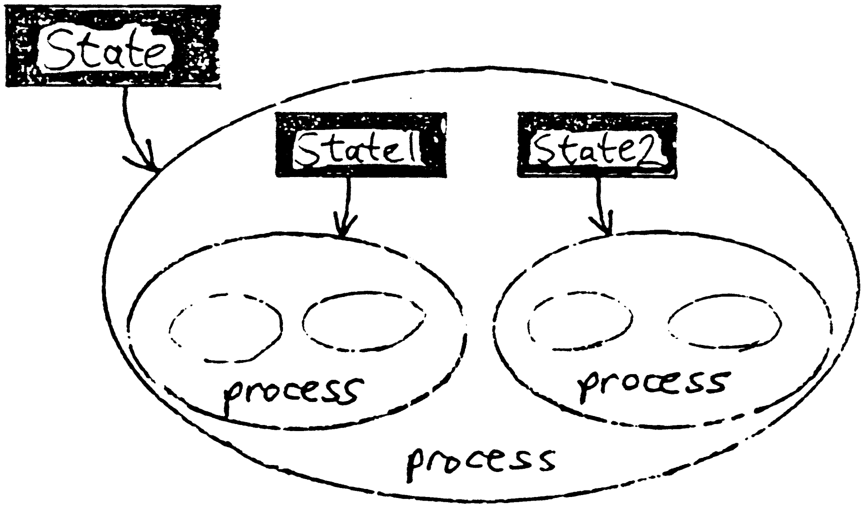


AND-PARALLELISM

Clause with >1 calls in body =
Creation of new processes

e.g

```
mode process(?).
process(State) ←
        process(State1), process(State2)
```



Unit clause =
Process termination

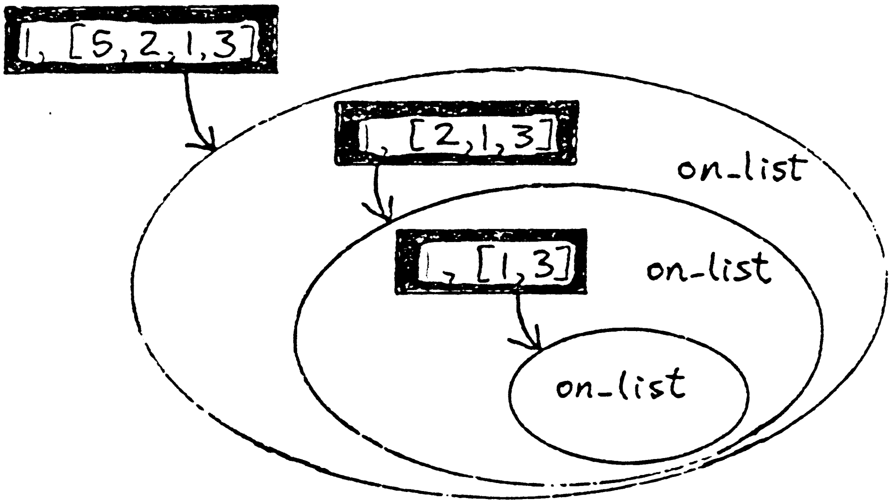
e.g

```
process(State).
```

AND-PARALLELISM

```
mode on_list(?,?).
on_list(Item, [Item|List]).
on_list(Item, [U|List]) ← Item \= U :
    on_list(Item, List).
```

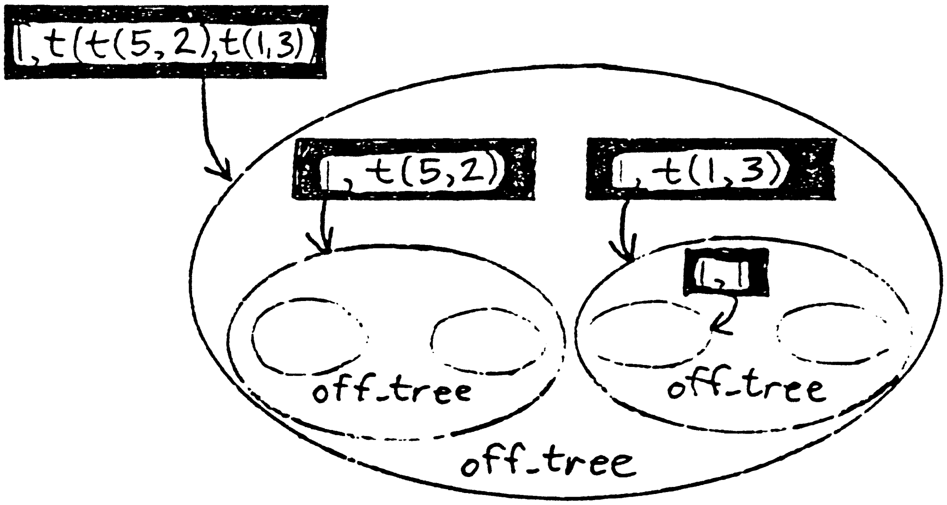
e.g. on_list(1,[5,2,1,3])



AND-PARALLELISM

```
mode off_tree(?,?).
off_tree(Item,t(T1,T2)) ←
    off_tree(Item,T1), off_tree(Item,T2).
off_tree(Item, L) ←
    integer(L),   Item \=L  :  true.
```

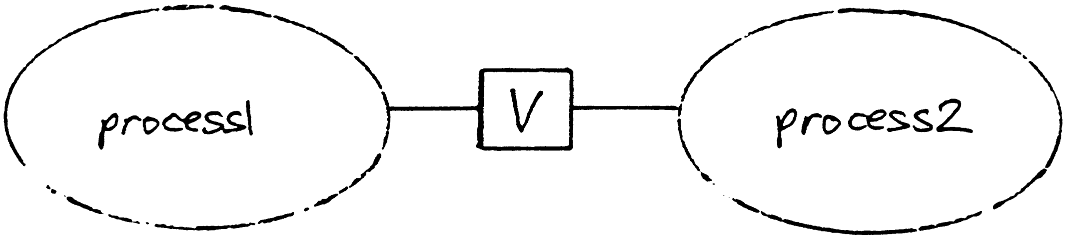
e.g. off_tree(1,t(t(5,2),t(1,3)))



COMMUNICATION

Shared variable = Communication channel or
memory location

e.g. process1(V), process2(V)



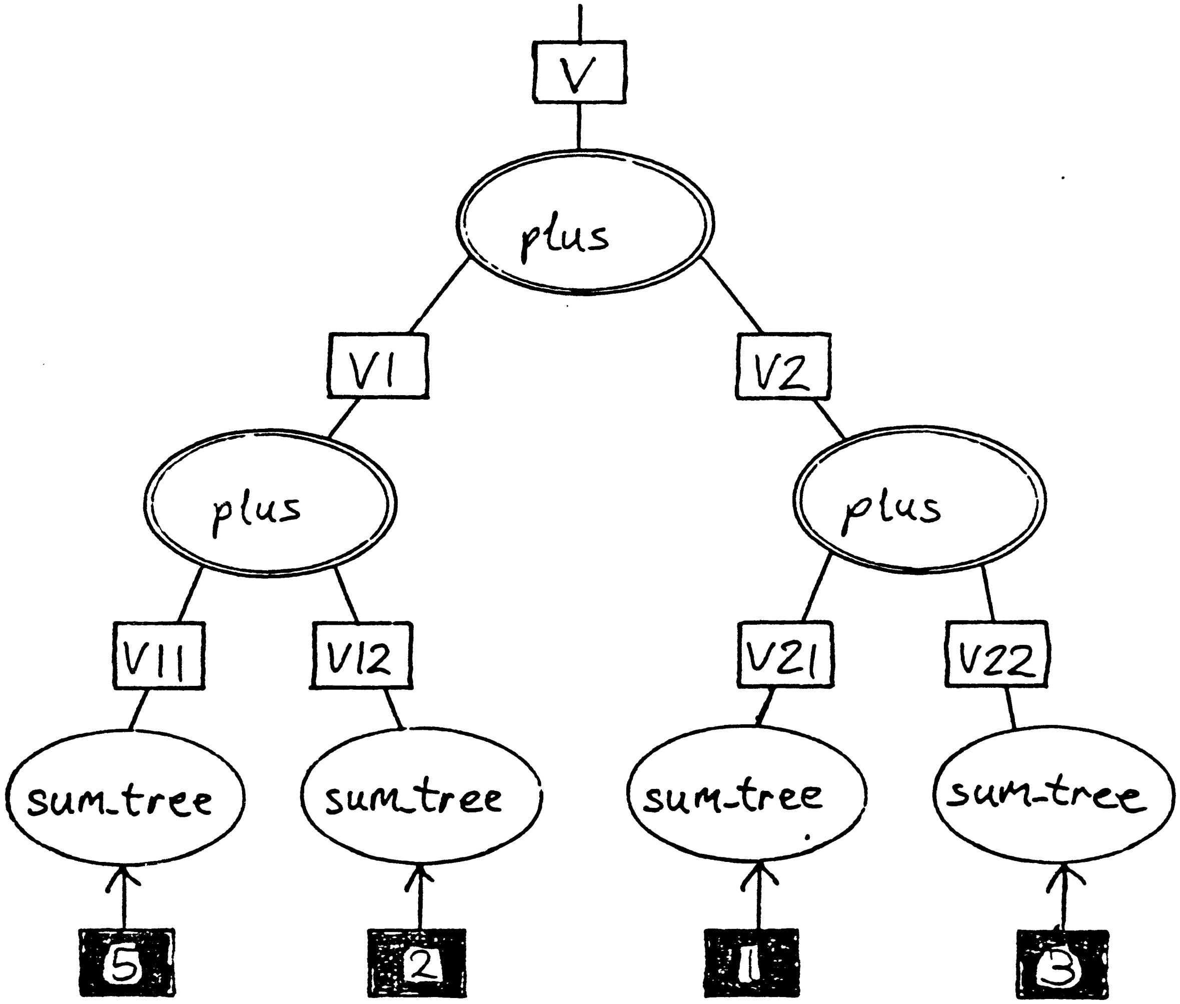
Binding shared variable = sending message

N.B. Single-assignment

COMMUNICATION

```
mode sum_tree(?,↑).
sum_tree(t(T1,T2),V) ←
    sum_tree(T1,V1), sum_tree(T2,V2),
    plus(V1,V2,V).
sum_tree(L,V) ← integer(L) :
    V=L.
```

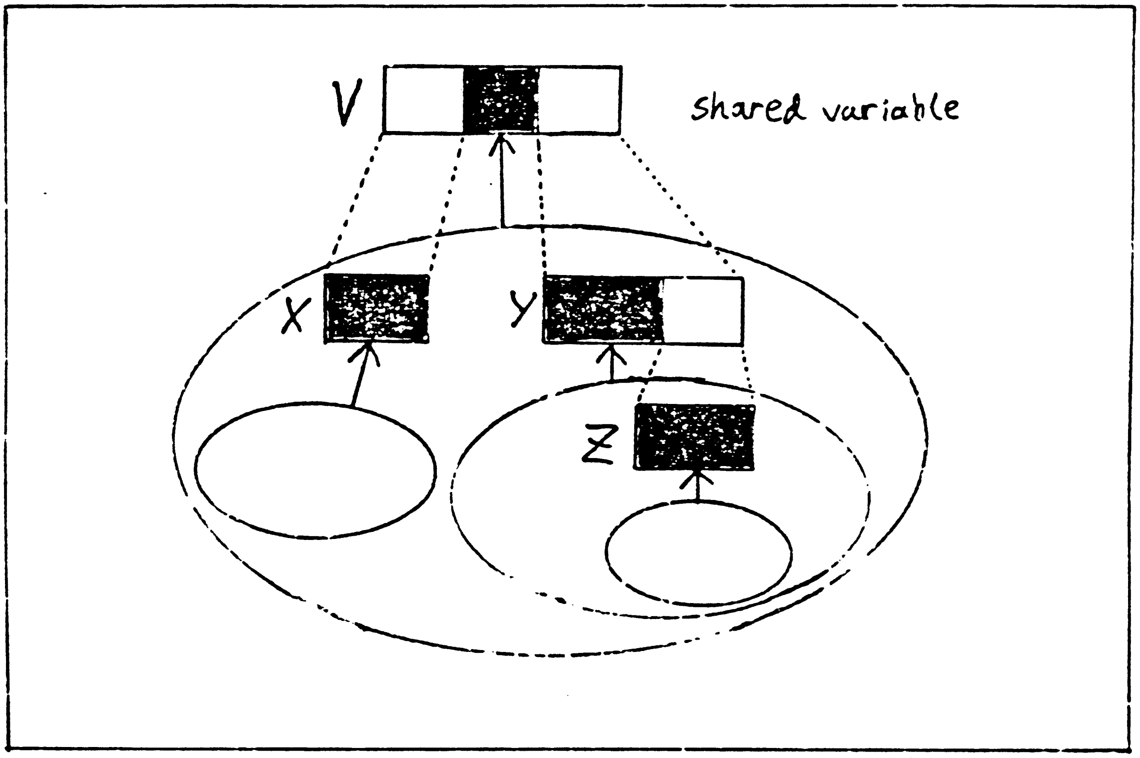
e.g. sum_tree(t(t(5,2),t(1,3)),V)



COMMUNICATION

Stream communication

Sequence of partial bindings to shared variable = Stream of messages

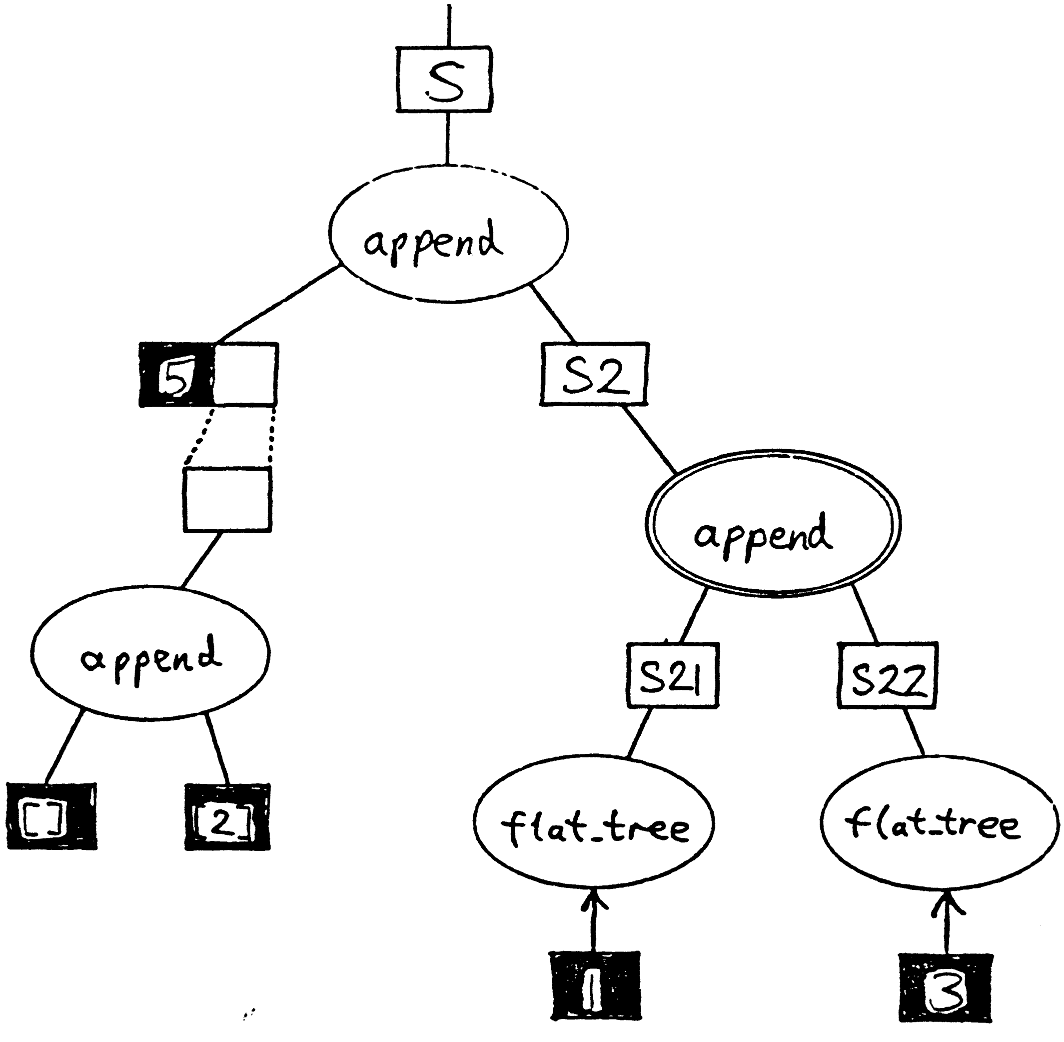


$V = f(X, Y) \quad X = t1 \quad Y = g(Z) \quad Z = t2$

COMMUNICATION

```
mode flat_tree(?,↑).
flat_tree(t(T1,T2), S) ←
    flat_tree(T1,S1), flat_tree(T2,S2),
    append(S1,S2,S).
flat_tree(L,[L]) ← integer(L) : true.

mode append(?,?,↑).
append([U|X],Y,[U|Z]) ← append(X,Y,Z).
append([],Y,Y).
```

e.g. flat_tree(t(t(5,2),t(1,3)), S)

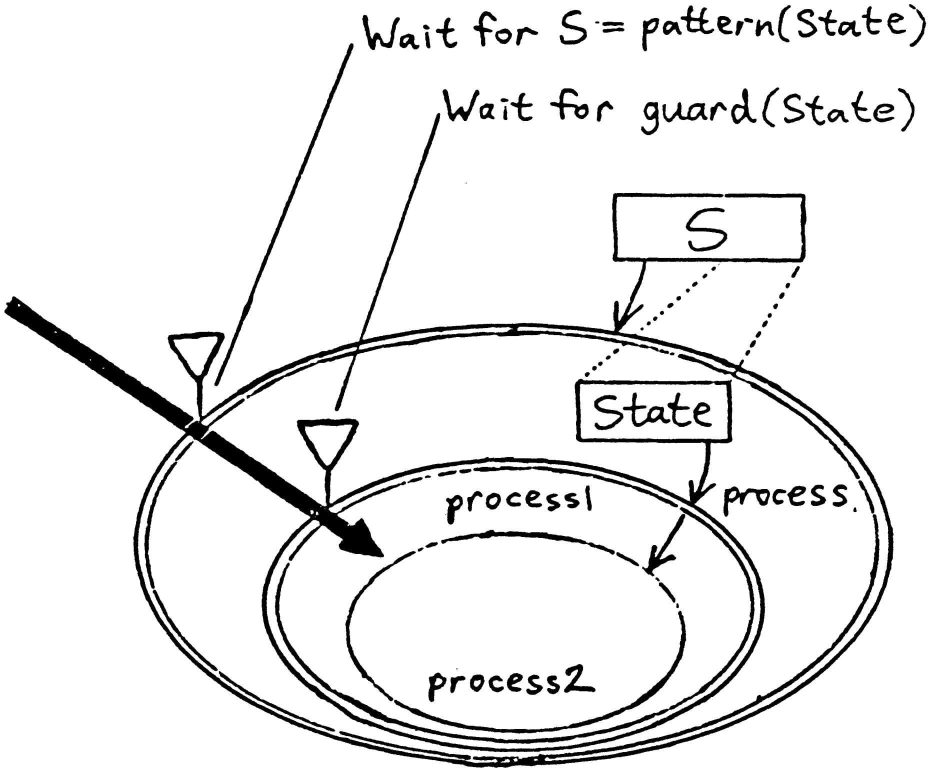


SYNCHRONIZATION

Delay reduction (call → clause body) until:

1. Input arguments available, and
2. Guard succeeds

```
mode process(?), process1(?).
process(pattern(State)) ←
    process1(State).

process1(State) ← guard(State) :
    process2(State).
```



L6

SYNCHRONIZATION

Before reduction

No access to output arguments of call.
Only input (read-only) access to input arguments of call.

1. One-way unification (matching)

Unification (call – clause head) cannot
bind call variables:
suspends on attempt to do so.

$$\begin{array}{ll} \text{Call} & r(t1,\ldots,tk) \\ & \quad\Downarrow \qquad \Downarrow \\ \text{Clause head} & r(t1',\ldots,tk') \end{array}$$

2. Safe guards

Guard cannot bind call variables.

After reduction

Call arguments are unified with head
arguments, in output positions.
May bind call variables.

SYNCHRONIZATION

Kernel PARLOG - make matching explicit

```
mode p(?,↑,?,↑).
p(t1,t2,t3,t4) ← guard : body.
```

can be written:

```
p(X1,X2,X3,X4) ← t1<=X1, t3<=X3, guard:
                  X2=t2, X4=t4, body.
```

Matching primitive

```
t1 <= t2
```

Unify t1, t2 but <u>suspend</u> on attempt to bind variables in t2.

SYNCHRONIZATION

1.
```
mode append(?, ?, ↑).

append([U|X], Y, [U|Z]) ← append(X, Y, Z).
append([], Y, Y).
```

⇓ PARLOG

Kernel PARLOG

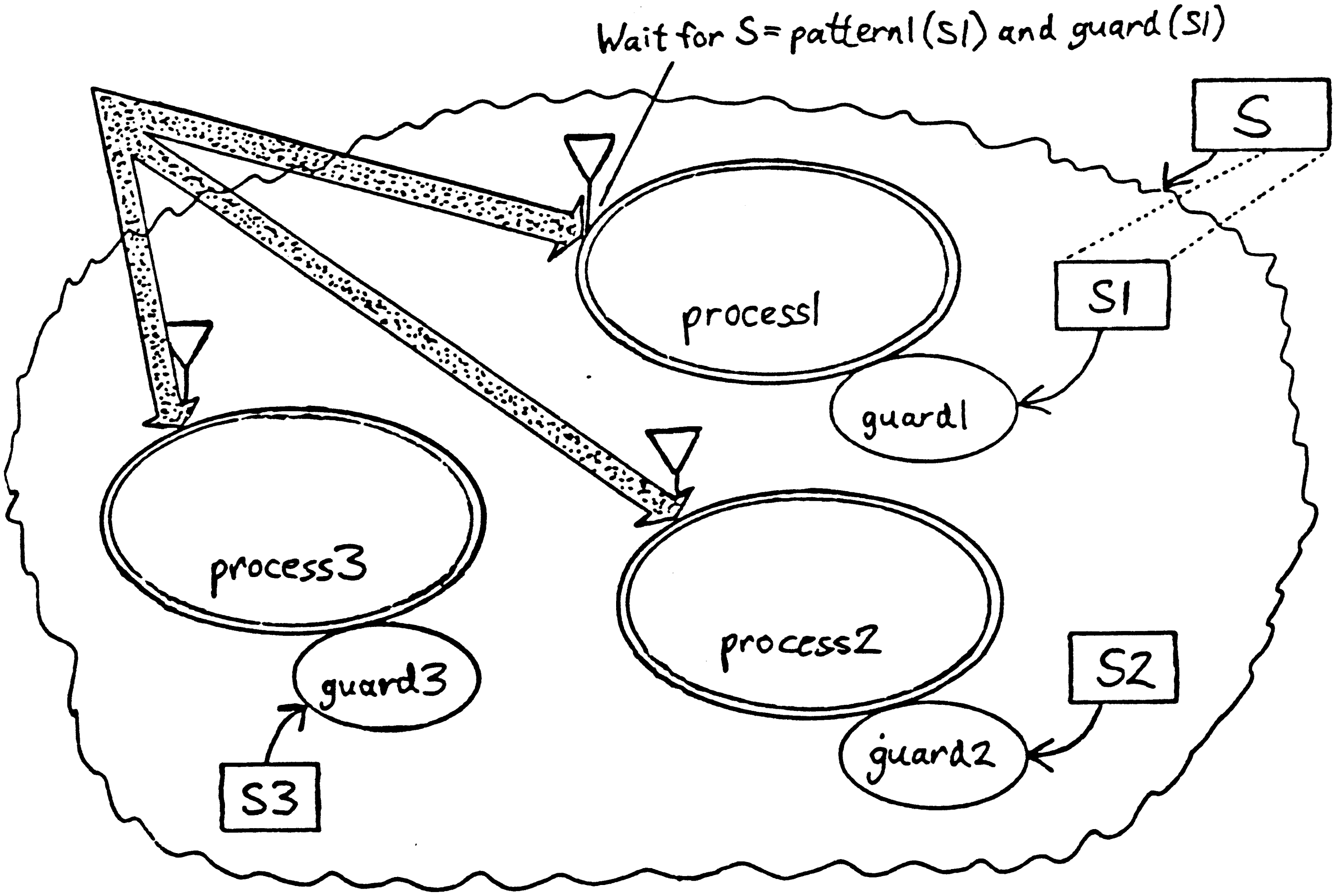
2.
```
append(T, Y, S) ← [U|X] <= T :
                  S = [U|Z], append(X, Y, Z).
append(T, Y, S) ← [] <= T :
                  S = Y.
```

COMMITTED CHOICE NON-DETERMINISM

Clauses in procedure = alternative ways to reduce process

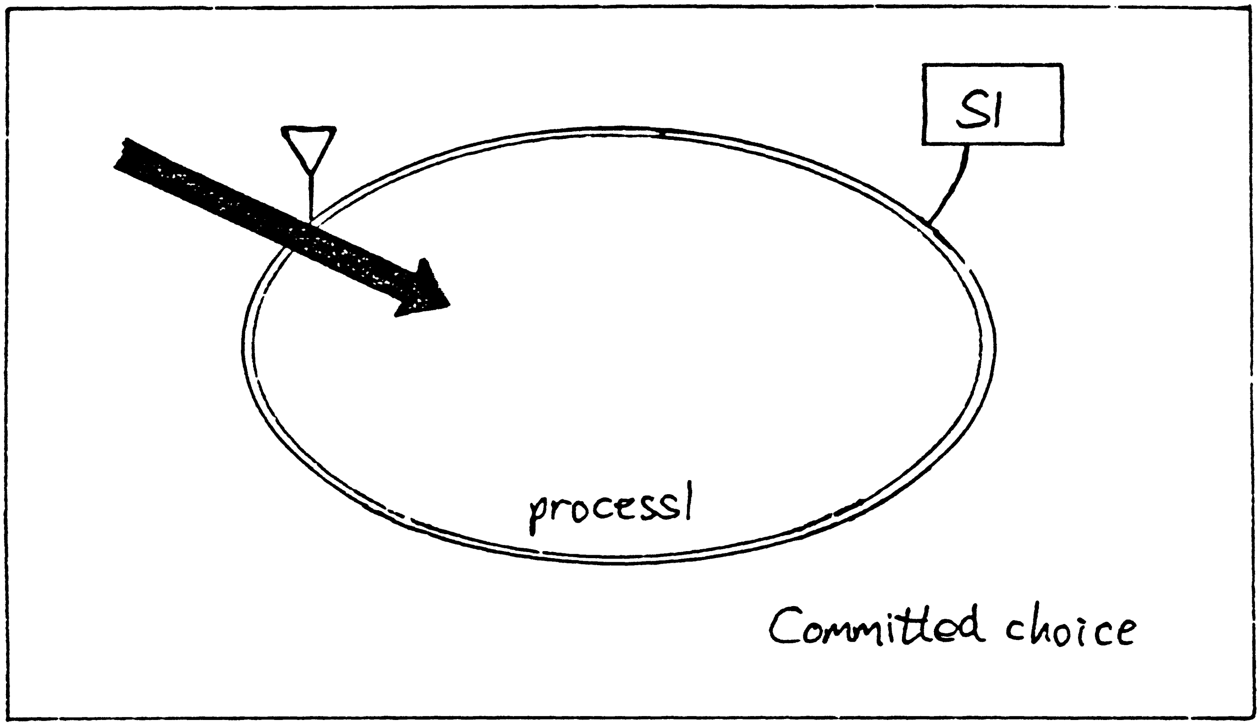
Committed choice of "candidate" clause
("candidate": successful input matching and guard)

```
mode process(?).
e.g. process(pattern1(S1)) ← guard1(S1) : process1(S1).
     process(pattern2(S2)) ← guard2(S2) : process2(S2).
     process(pattern3(S3)) ← guard3(S3) : process3(S3).
```



Parallel search

COMMITTED CHOICE NON-DETERMINISM



Read-only access to arguments during search
    – "safe" guards.

Output only after "commitment".
    – committed to output.

May be many candidate clauses.
    – time dependency.

Parallel search for clause.
    – "committed" or-parallelism.

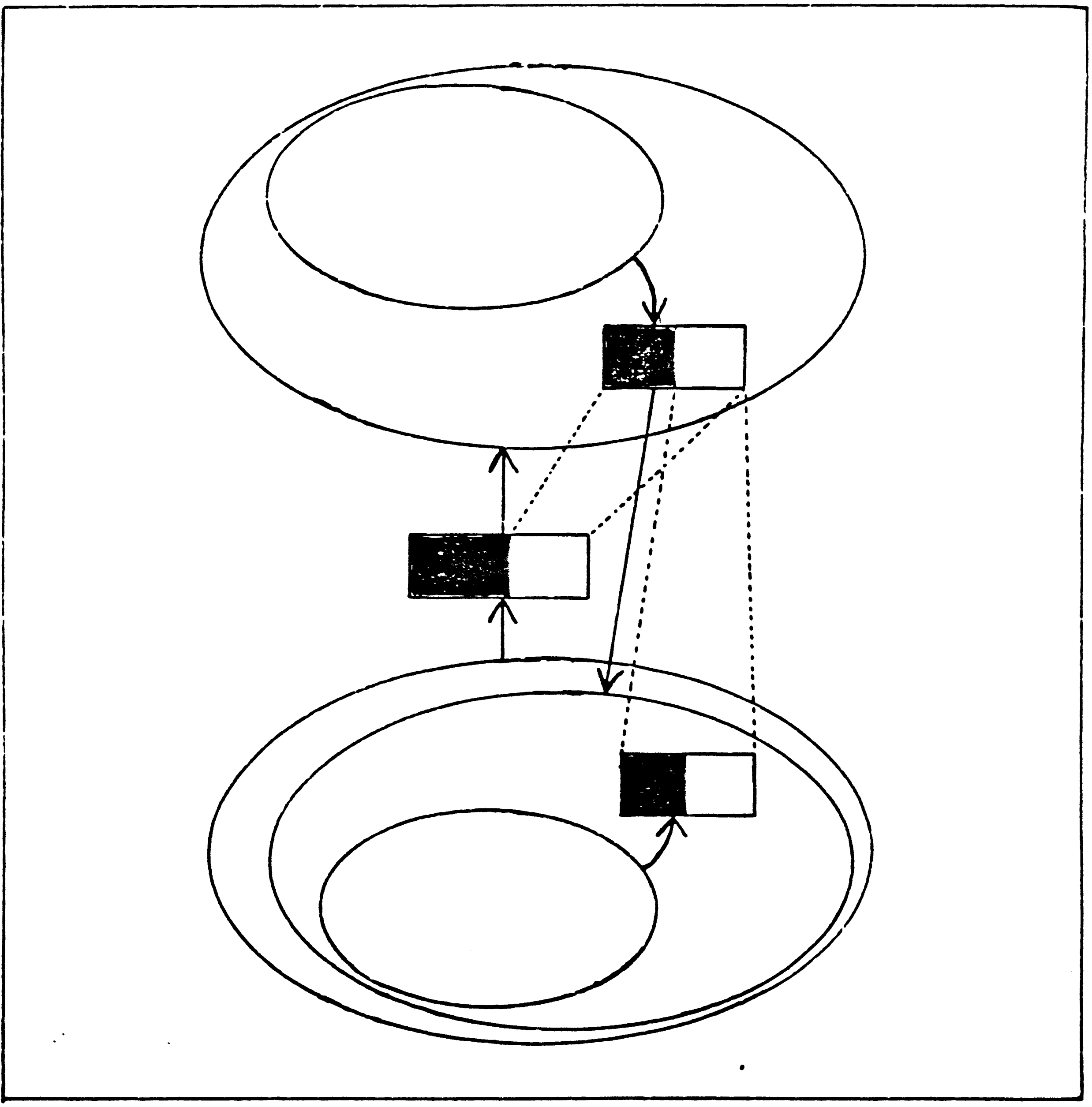
COMMITTED CHOICE NON-DETERMINISM

Committed or-parallelism

```
mode on_tree(?,?).
on_tree(Item, t(T1,T2)) <- on_tree(Item, T1):
                           true.

on_tree(Item, t(T1,T2)) <- on_tree(Item, T2):
                           true.

on_tree(Item, Item).
```

THE LOGICAL VARIABLE

Sequence of partial bindings to shared variable can be made by different processes.



Cooperative construction of data.

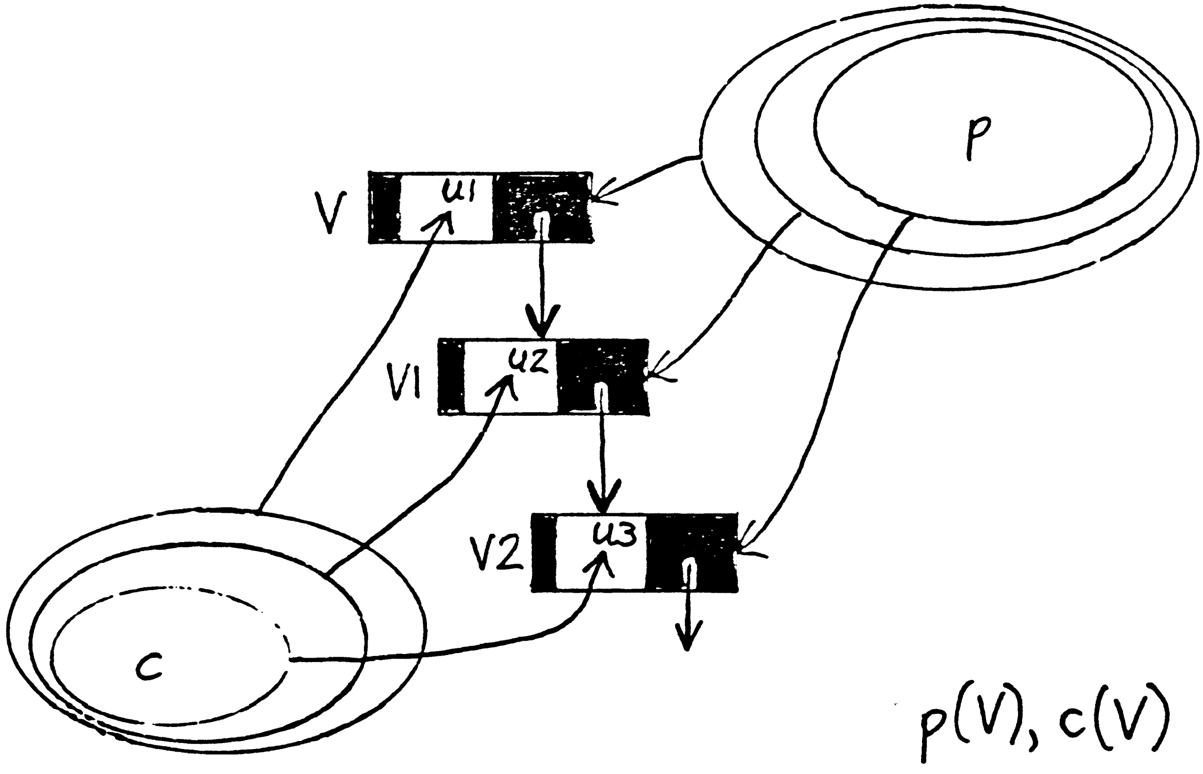
THE LOGICAL VARIABLE

Back communication

Sequence of partial bindings to shared variable, some by this process, some by others.

e.g. p: $V = [t(U1) | V1]$ $V1 = [t(U2) | V2]$

c: $U1 = t1$ $U2 = t2$



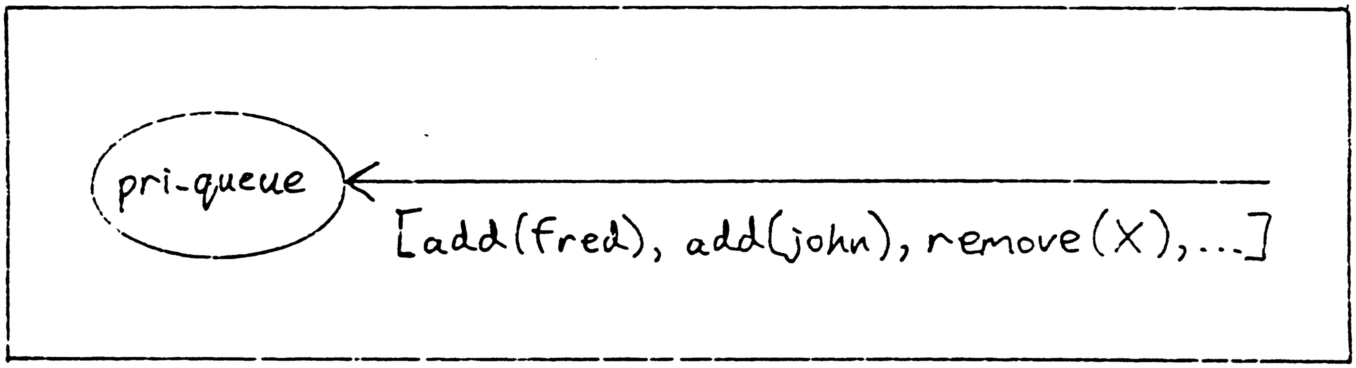
p(V), c(V)

Back communication + merge = monitors

Back communication → lazy (demand-driven) evaluation

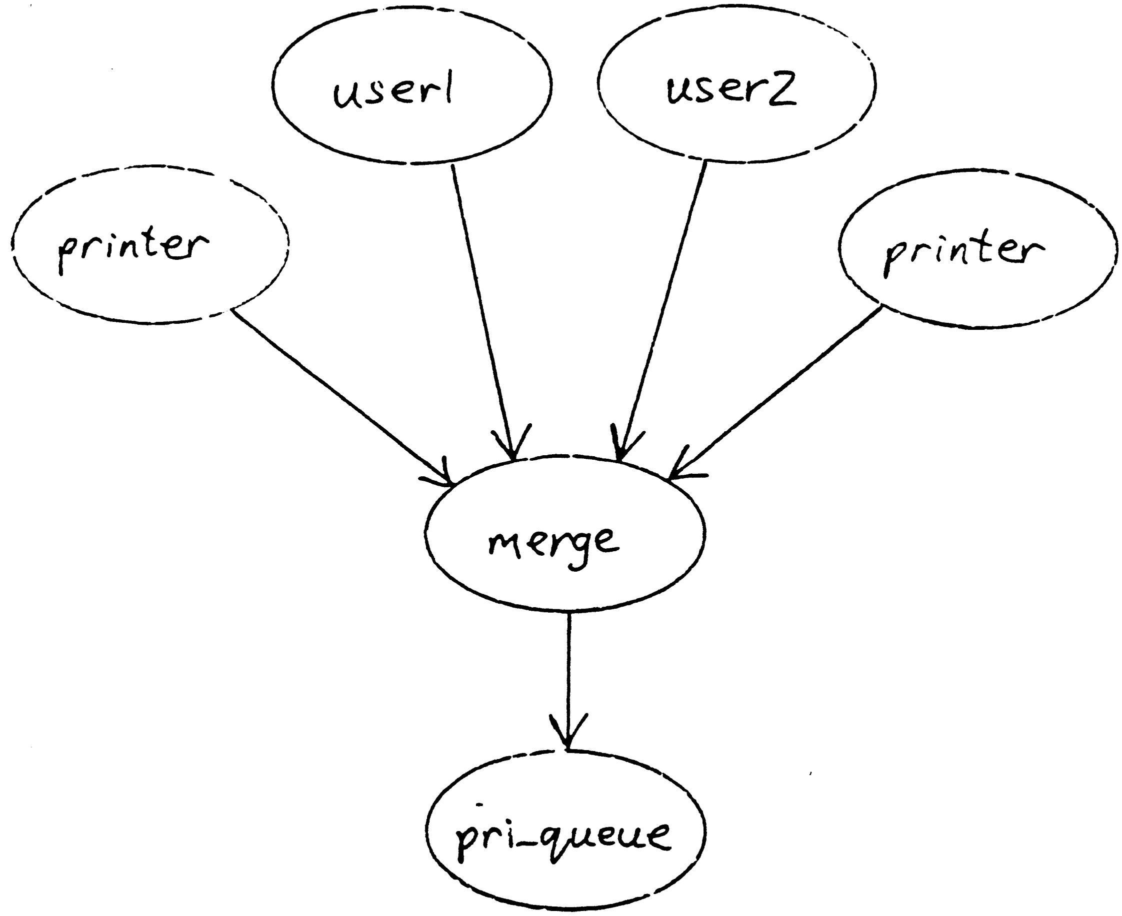
THE LOGICAL VARIABLE

```
mode pri-queue(?), pri-queue(?,?).
pri_queue(M)  ←  pri-queue(M, []).

pri-queue([add(Item)|M], Q)  ←
     insert(Item, Q, New_q),
     pri-queue(M, New_q).
pri_queue([remove(Item)|M], [H|Q])  ←
     Item = H,
     pri-queue(M, Q).
```



THE LOGICAL VARIABLE

e.g. priority spooler



THE LOGICAL VARIABLE

Eager "read list":

```
mode eager-read(↑).

eager-read([]) ← end-of-file : true ;
eager-read([U|X]) ←
    read(U) &
    eager-read(X).
```

Lazy "read list":

```
mode lazy-read(?).
lazy-read([U|X]) ← end-of-file :
    U = end-of-file ;
lazy-read([U|X]) ←
    read(U) &
    lazy-read(X).
```

METALEVEL PROGRAMMING

Problem:

To allow a process ("metaprogram") to examine and control evaluation of another ("object program").

Solution 1:

Transform the object program to signal its status and respond to control messages.

Solution 2:

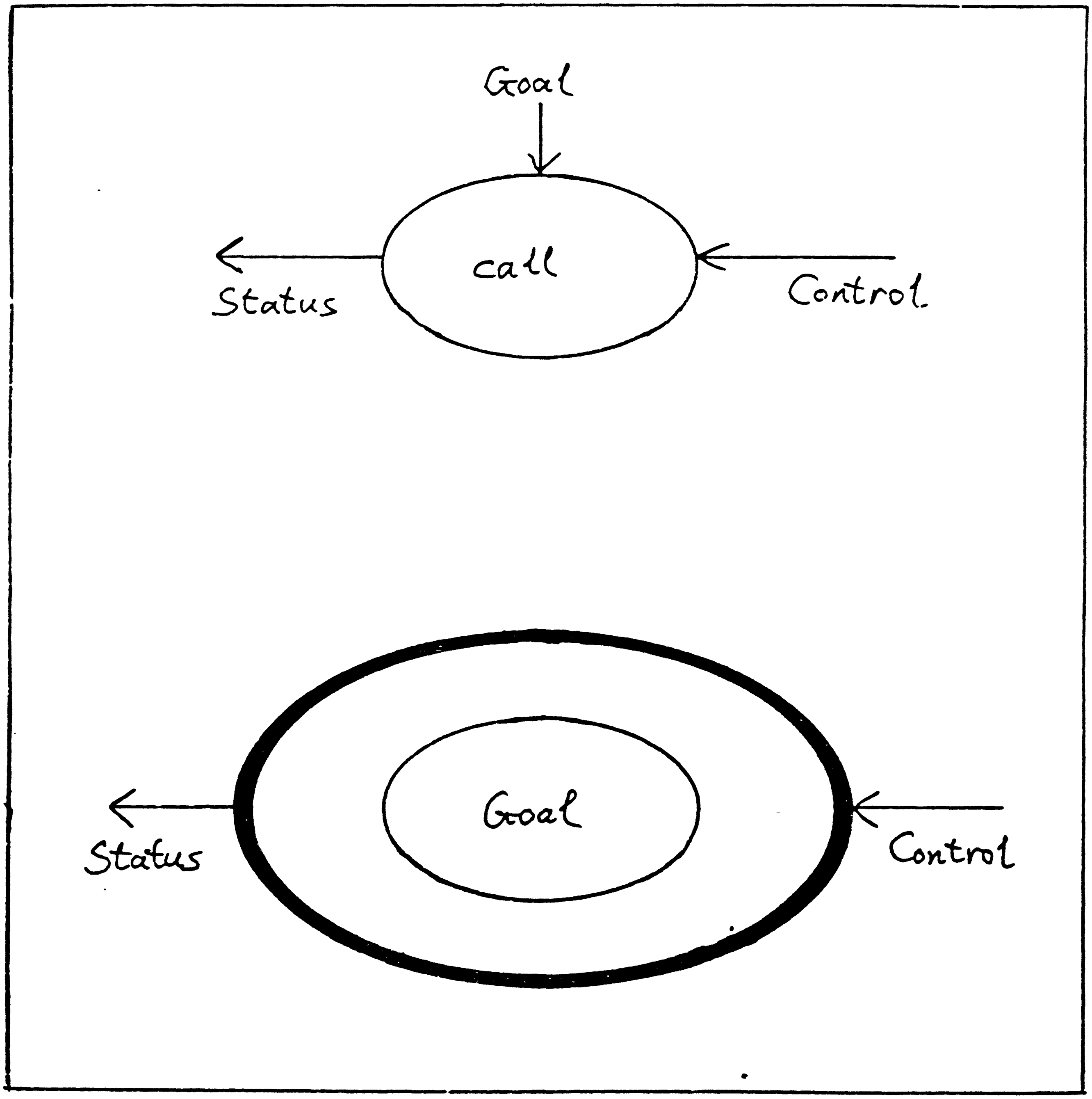
Encapsulate object program in special metacall to achieve same effect.

C17

METALEVEL PROGRAMMING

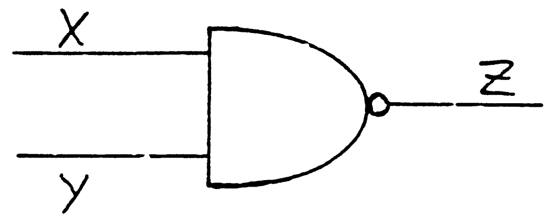
The PARLOG metacall

call(Goal?, Status↑, Control?)



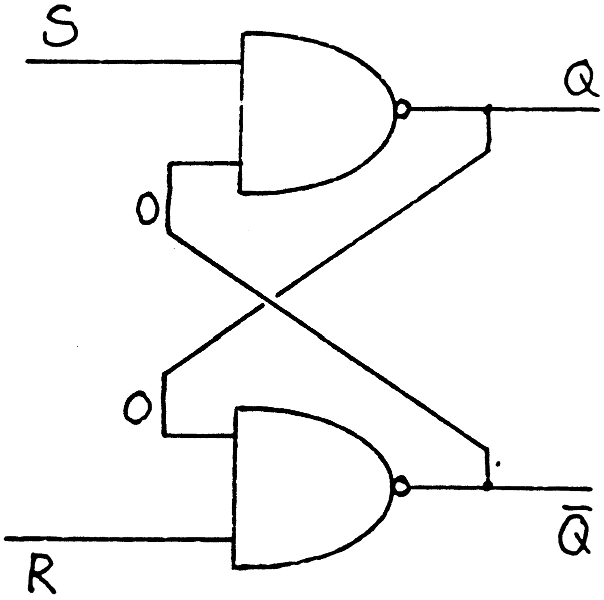
EXAMPLE: SR LATCH

PARLOG specification of NAND gate



```
mode nand(?,?,↑).

nand([1|X],[1|Y],[0|Z]) ← nand(X,Y,Z).
nand([0|X],[V|Y],[1|Z]) ← nand(X,Y,Z).
nand([U|X],[0|Y],[1|Z]) ← nand(X,Y,Z).
```

SR latch using NAND gates



```
mode sr_latch(?,?,↑,↑).

sr_latch(S,R,Q,Q_) ←
```

EXAMPLE: PARSER

Grammar

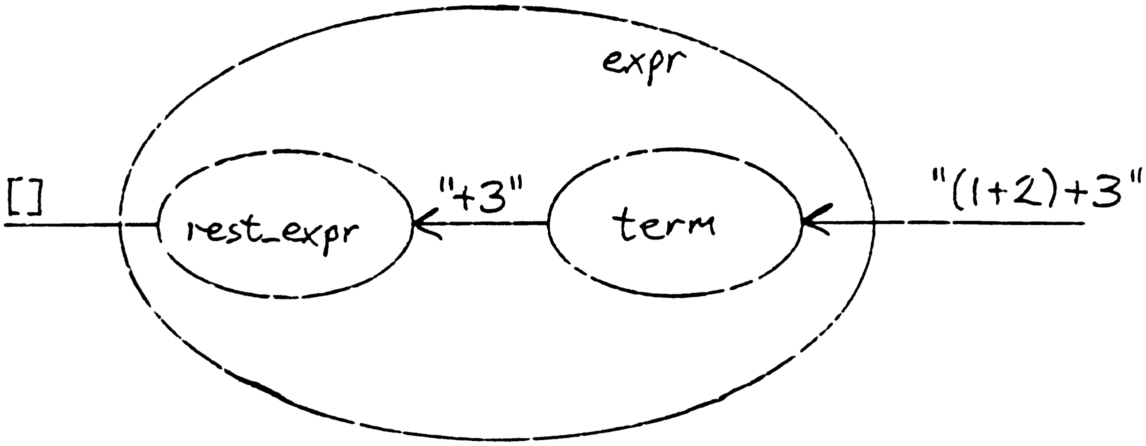
```
Expr  ->  Term Rest_expr

Rest_expr  ->  Add_op Expr
Rest_expr  ->  empty

Term  ->  Number
Term  ->  '(' Expr ')'
```

PARLOG

```
mode expr(?,↑), rest_expr(?,↑), term(?,↑).

expr(Tokens_h, Tokens_t) ←
    term(Tokens_h, Tokens), rest_expr(Tokens, Tokens_t).

rest_expr([Op|Tokens_h], Tokens_t) ← add_op(Op) :
    expr(Tokens_h, Tokens_t) ;
rest_expr(Tokens, Tokens).

term([N|Tokens], Tokens) ← number(N) : true.
term(['('|Tokens_h], Tokens_t) ←
    expr(Tokens_h, [')'|Tokens_t]).
```

EXAMPLE: PARSER

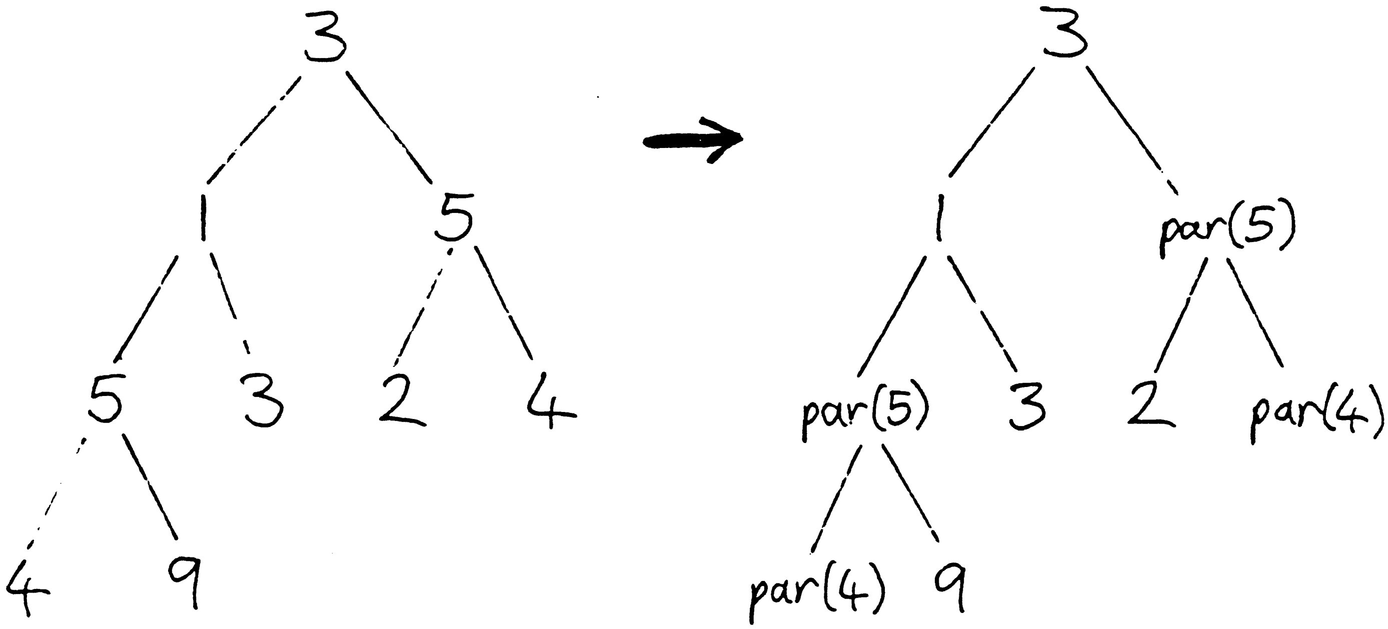
Process interpretation



EXAMPLE: LABEL TREE

Problem:

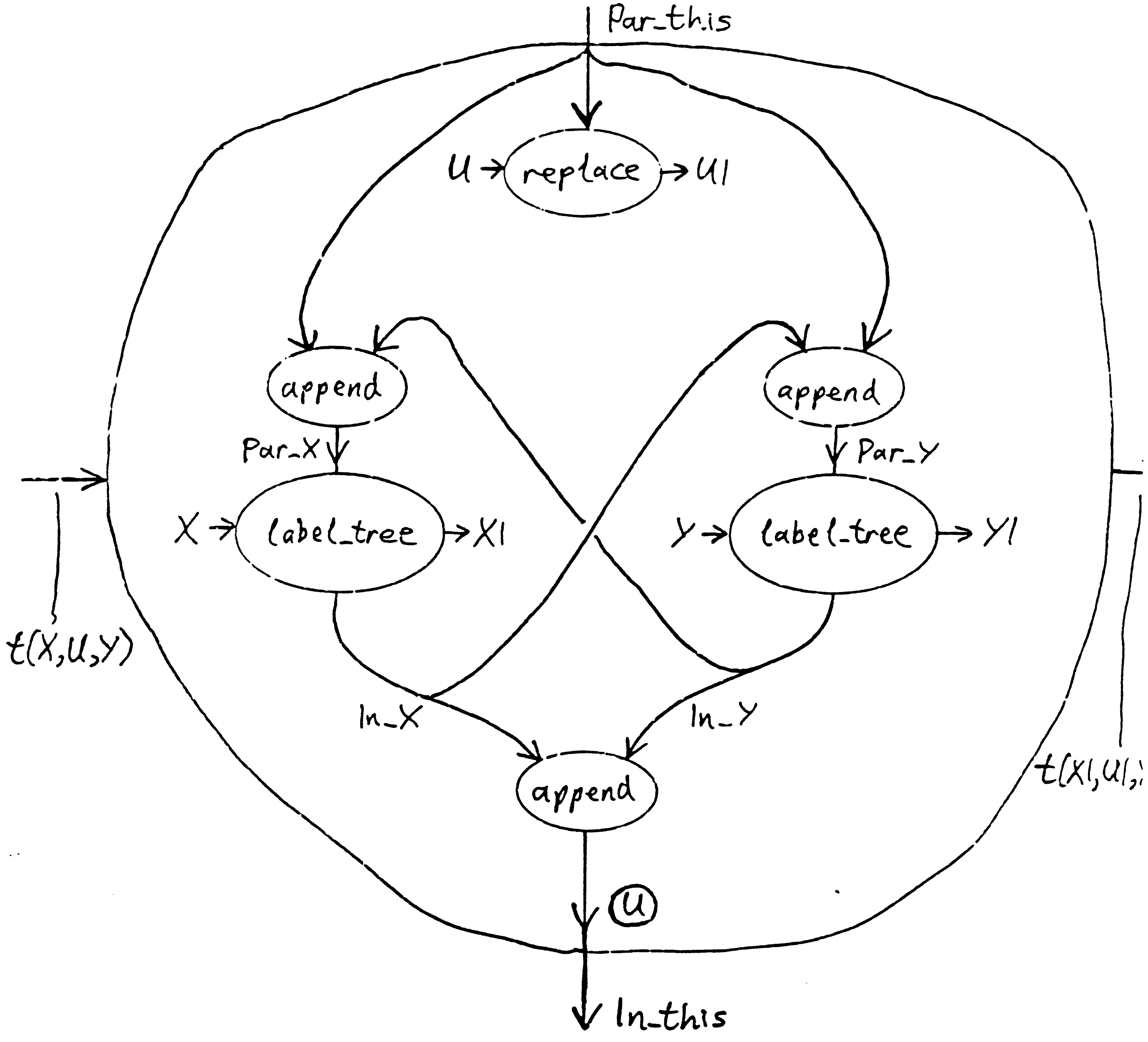
Mark nodes of a labelled binary tree that occur on parallel branches.



EXAMPLE: LABEL TREE

```
mode label_tree(I_tree?, O_tree↑, Par_this?, In_this↑).

label_tree(t(X,U,Y), t(X1,U1,Y1), Par_this, [U|In_this]) ←
    replace(U,U1, Par_this),
    label_tree(X,X1, Par_X, In_X),
    label_tree(Y,Y1, Par_Y, In_Y),
    append(Par_this, In_Y, Par_X),
    append(Par_this, In_X, Par_Y),
    append(In_X, In_Y, In_this).
label_tree(empty, empty, Par_this, []).


mode replace(Node?, New_node↑, Par_nodes?).

replace(Node, par(Node), Par_nodes) ←
    on_list(Node, Par_nodes) : true ;
replace(Node, Node, Par_nodes).
```

EXAMPLE: LABEL TREE



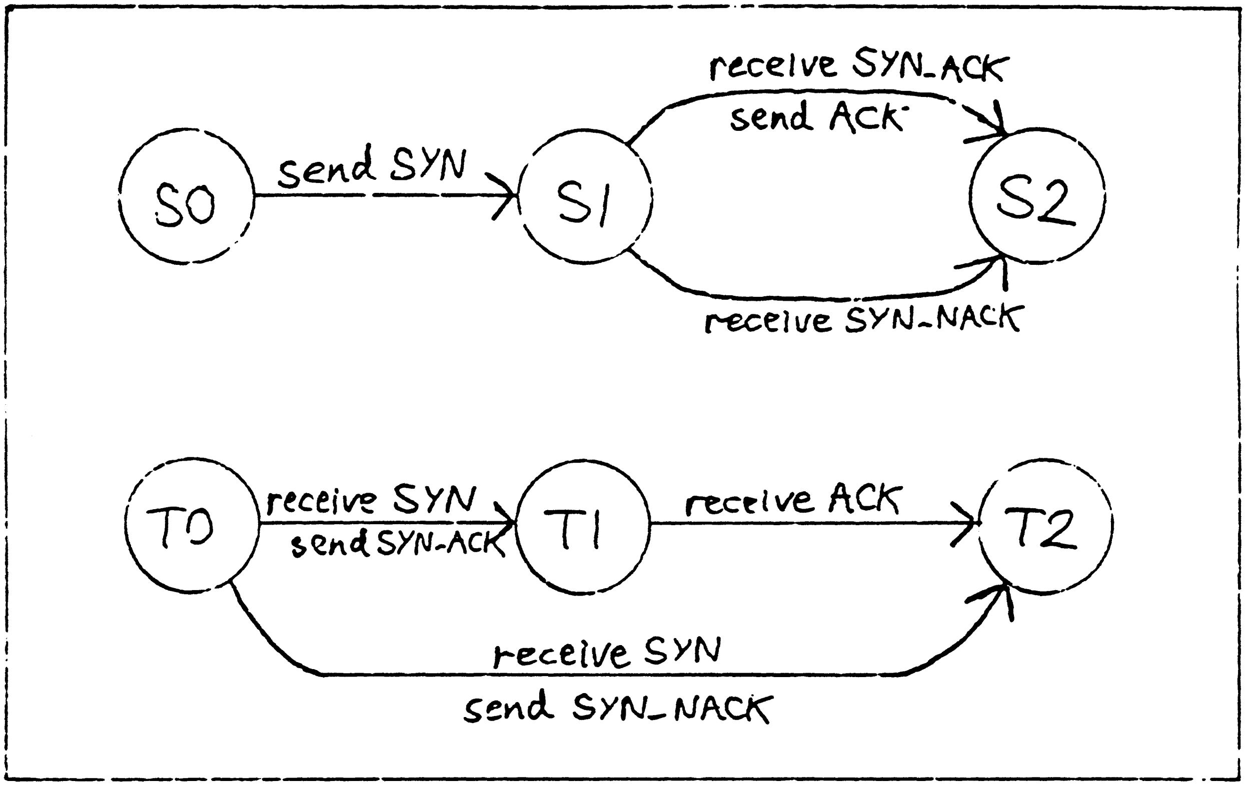
EXAMPLE: COMMUNICATION PROTOCOLS

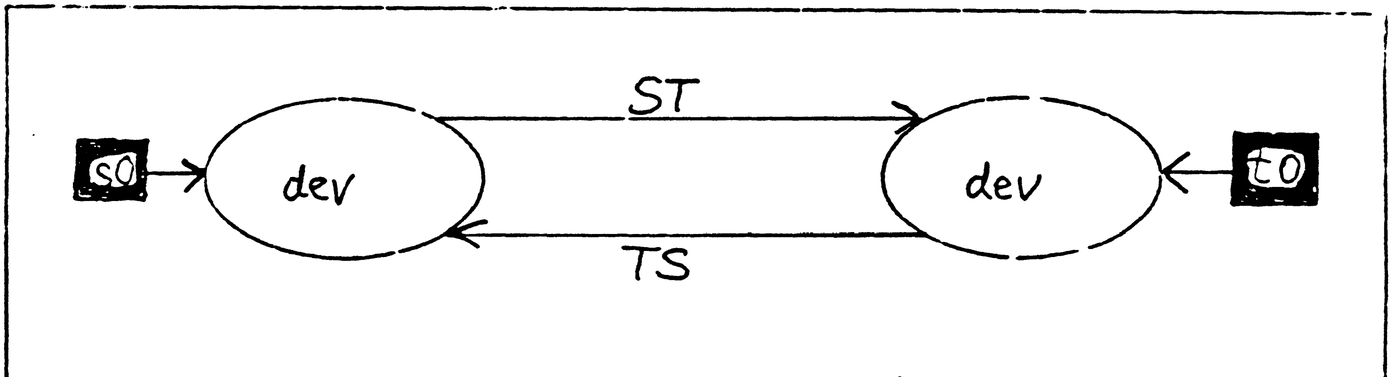
```
mode dev(State?, In?, Out↑).

dev(s0, In, [syn|Out]) ← dev(s1, In, Out).
                                    % Establish

dev(t0, [syn|In], [syn_ack|Out]) ← dev(t1, In, Out).
                                    % Acknowledge
dev(t0, [syn|In], [syn_nack|Out]) ← dev(t2, In, Out).
                                    % Decline

dev(s1, [syn_ack|In], [ack|Out]) ← dev(s2, In, Out).
                                    % Confirm
dev(s1, [syn_nack|In], Out) ← dev(s2, In, Out).
                                    % Abandon

dev(t1, [ack|In], Out) ← dev(t2, In, Out).
                                    % Connect
```

7

EXAMPLE: COMMUNICATION PROTOCOLS

PARLOG specification of simple connection establishment protocol





dev(s0, TS, ST), dev(t0, ST, TS)

EXAMPLE: PARLOG FOR SPECIFICATION

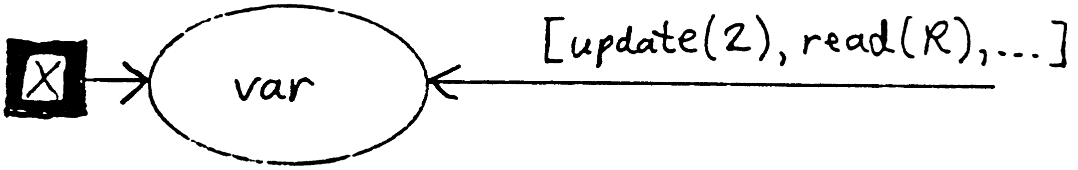
CSP specification of (illogical?) variable

$$\begin{aligned} var_X = (\, & update?Y \rightarrow var_Y \mid \\ & read!X \rightarrow var_X \,) \end{aligned}$$

Advantage: calculus for reasoning about behaviour

PARLOG specification

```
mode var(?,?).
var(X, [update(Y)|M]) <- var(Y,M).
var(X, [read(R)|M]) <- R=X, var(X,M).
var(X, []).
```

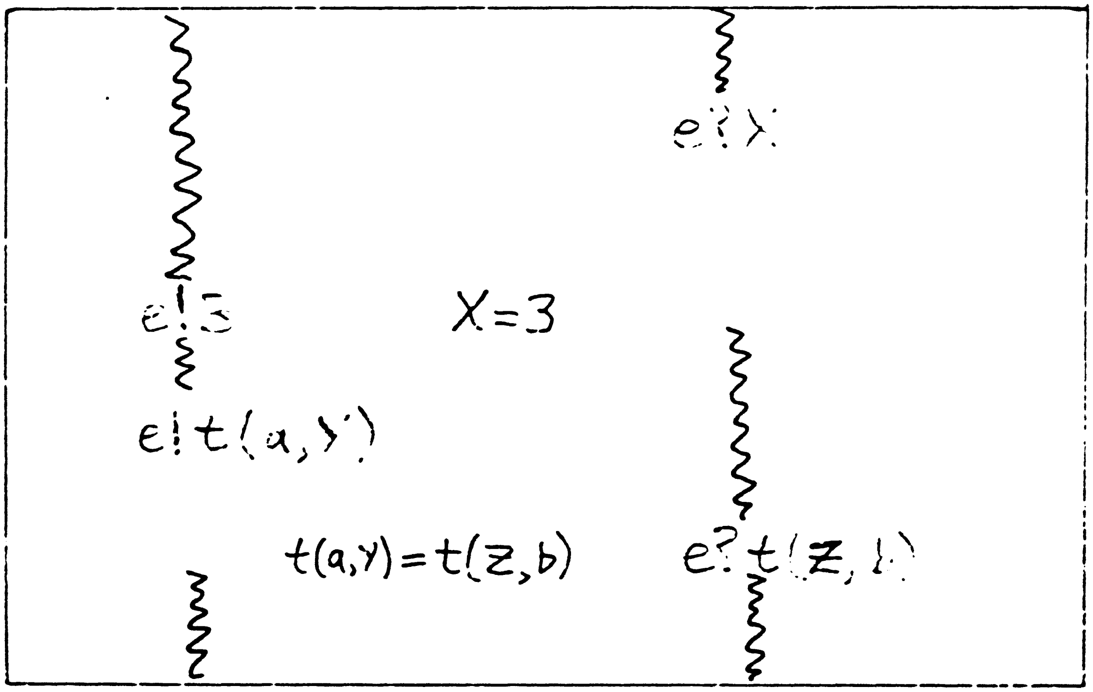


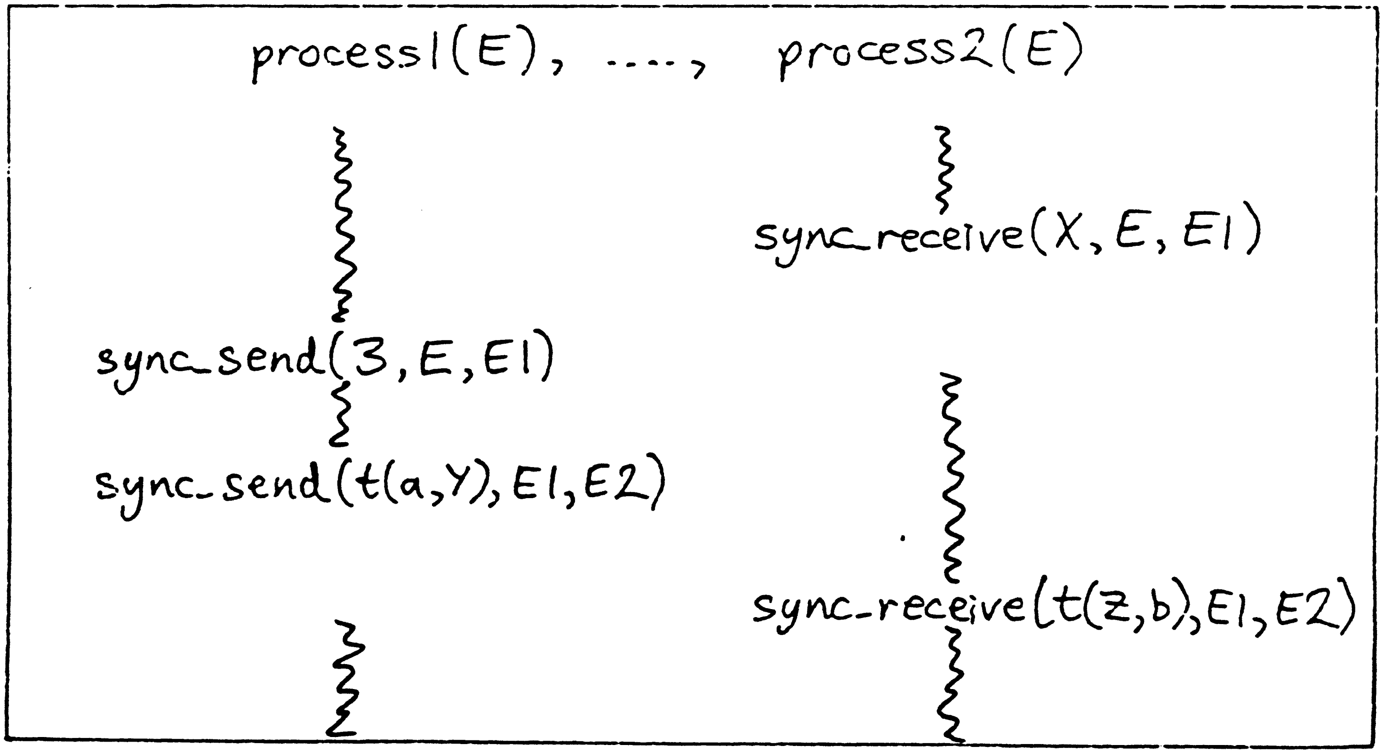
Advantage: some properties clear from logical reading

EXAMPLE: SYNCHRONIZED COMMUNICATION

```
mode sync_send(?,↑,?), sync_receive(?,?,↑),
     succeeded(?).

sync_send(Term,[(Term,Ack)|Ch],Ch) ←
    succeeded(Ack).

succeeded(succeeded).

sync_receive(R_term,[(Term,Ack)|Ch],Ch) ←
    R_term = Term &
    Ack = succeeded.
```

EXAMPLE: SYNCHRONIZED COMMUNICATION





Page 38 missing: more of "Example: Synchronized Communication"?

Concurrent Prolog Differences

Synchronization: read-only variable

- Read-only annotation ('?') on consumer occurrences of variables.
- Suspend on attempt to bind a variable via a read-only occurrence.
- No other restrictions on variable bindings made by unification or guards; call variables may be bound before commitment

  => multiple environments

  e.g.
  ```
  append([U|X],Y,[U|Z]) :- append(X?,Y?,Z).
  append([],Y,Y).
  ```

  "mode" depends on annotations in call.
- Difficulties implementing multiple environments

  => serious implementation work thus far restricted to Flat Concurrent Prolog (FCP)
- Flat: only calls to system primitives in guards

GHC Differences

Syntax

Assumed mode (?, ..., ?).

Output unification done explicitly.

e.g. PARLOG

```
mode append(?,?,^).
append([U|X],Y,[U|Z]) <- append(X,Y,Z).
append([],Y,Y).
```

GHC

```
append([U|X],Y,Z1) :- Z1 = [U|Z],
    append(X,Y,Z).
append([],Y,Z1) :- Z1 = Y.
```

Synchronization

1. One-way unification (matching):

   Call/head unification cannot bind call variables

   (like PARLOG).

2. Guard **suspends** on attempt to bind call variables

   (PARLOG: guard must be "safe").

N.B. Flat GHC = Flat PARLOG.

Applications of Parallel Logic Languages

Why use parallel logic languages?

- Parallel logic languages =

  Horn clause logic + concurrent, committed-choice
  proof procedure

- Horn clause logic: (declarative content)

  parallel logic programs can be read as executable specifications

- Proof procedure: (parallel execution)

  declarative programs can be executed in AND-parallel

- Proof procedure: (process interpretation)

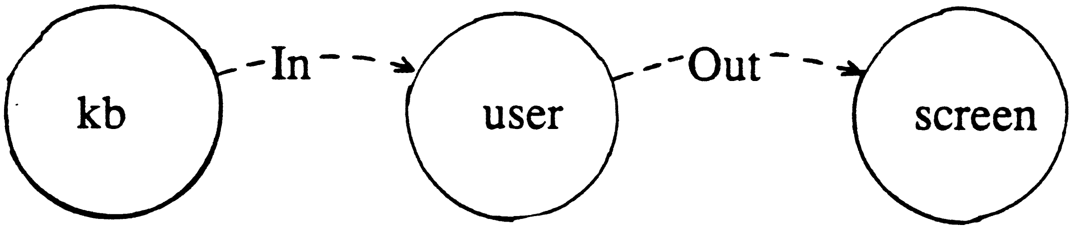
  can be exploited to provide useful operational behaviour

Useful Operational Behaviour

The proof procedure of parallel logic languages permits them to implement many useful 'real-world' behaviours

- Concurrent, communicating entities

```
..., keyboard(In), user(In, Out), screen(Out), ...
```

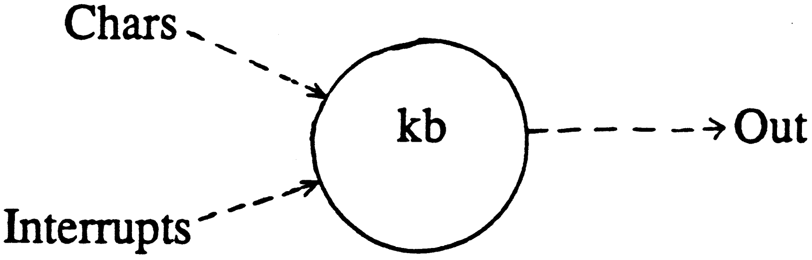

kb
In
user
Out
screen

- Side-effects in the external world

```
screen([ display(X) | In]) ← output(X) & screen(In).
```

- Time dependent treatment of events:

```
keyboard([Ch | Chars], Interrupts, Out) ←
        handle_char(Ch,Out,NewOut),
        keyboard(Chars, Interrupts, NewOut).
keyboard(Chars, [Int | Interrupts] Out) ←
        handle_interrupt(Int, Out, NewOut),
        keyboard(Chars, Interrupts, NewOut).
```


Chars
kb
Out
Interrupts

Applications of Parallel Logic Languages
(Some Examples)

(a) **Behavioural**: describe, implement concurrent systems

(emphasis on operational behaviour)

operating systems / programming environments
telephone exchange control
simulation

(b) **Algorithmic**: describe, implement parallel algorithms

(emphasis on declarative content)

parallel parsers
parallel router
compilers
image processing
qualitative reasoning

(c) **Language Implementation:**

concurrent implementations of other language formalisms

Vulcan	)	Concurrent object-oriented languages
POLKA	)	
LOTOS		Formal description language for concurrent systems

A Large Application:
PARLOG Programming System (PPS)

- An operating system designed to support logic programming on parallel machines
- A prototype implemented on SUN workstations is a major PARLOG application
- Uses some UNIX facilities but implements computation control, secondary storage management etc in PARLOG
- Implementation exploits extended PARLOG metacall:
  - an exception handling mechanism
  - modularity
  - computation priorities
- Implementation demonstrates:
  - Use of PARLOG to implement a complex system
  - PARLOG programs as executable specifications
  - Use of PARLOG control metacall
  - PARLOG programming techniques:
    back communication, synchronization variables

PPS Facilities

PPS provides:

- modularity: programs are divided into databases

- persistence: file system is invisible
  databases persist between PPS invocations

- declarative environment:
  user view: system = {databases}
  user interaction: execute queries wrt databases
  queries calculate relations over system states

- metaprogramming:
  programs can:
  - reason about other programs
  - generate new versions of other programs

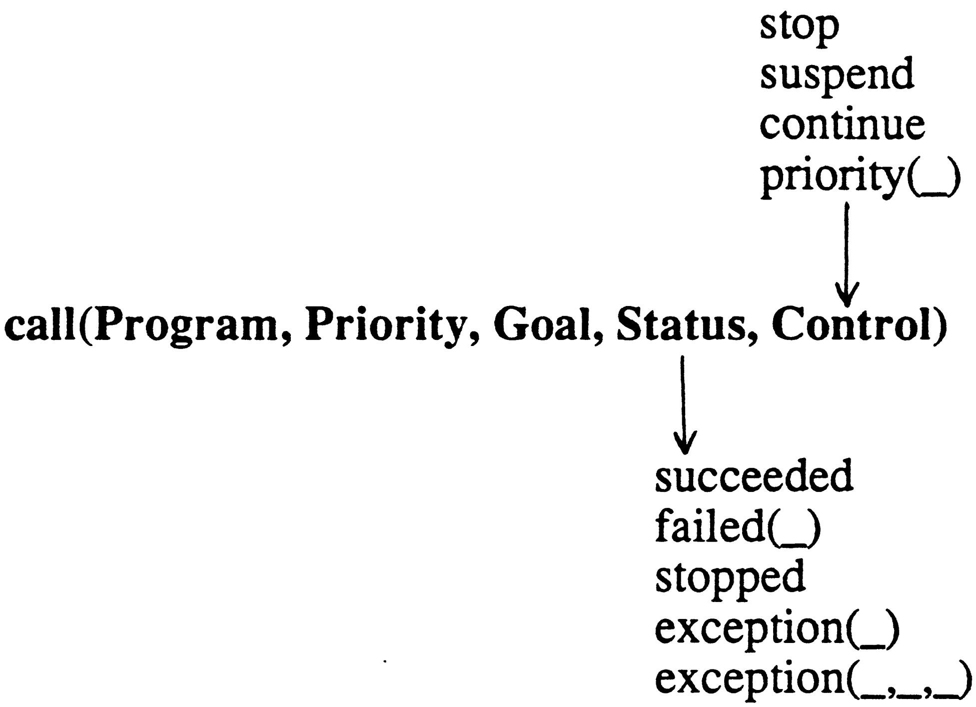
  programs define relations over system states.

- multiprogramming:
  task control
  concurrency control

- user-definable inference mechanisms:
  inheritance
  query-the-user

An Extended PARLOG Metacall

- Primitive for describing initiation, monitoring and control of a computation
- Efficient implementation techniques developed



stop
suspend
continue
priority(_)

call(Program, Priority, Goal, Status, Control)

succeeded
failed(_)
stopped
exception(_)
exception(_,_,_)

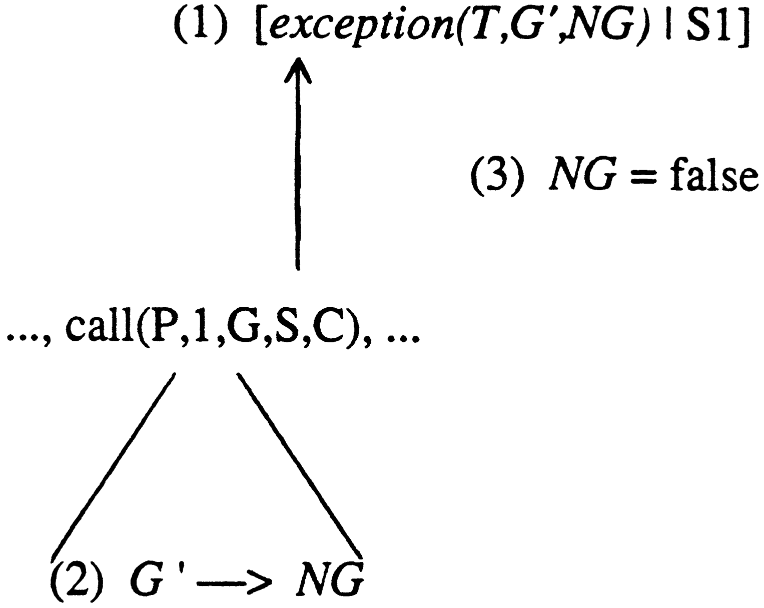
Program: names a module

Priority: programmer control of underlying scheduling mechanism

Status and **Control**: interfaces to monitoring and control functions of underlying machine

Exception Handling

(1) The *exception(T,G,NG)* message indicates an unsolvable goal *G*; *T* indicates why the goal was unsolvable
(undefined, div-by-zero, ...)



(1) [*exception(T,G',NG)* | S1]

(3) *NG* = false

..., call(P,1,G,S,C), ...

(2) *G* ' —> *NG*

(2) The unsolvable goal is replaced with a continuation variable *NG*

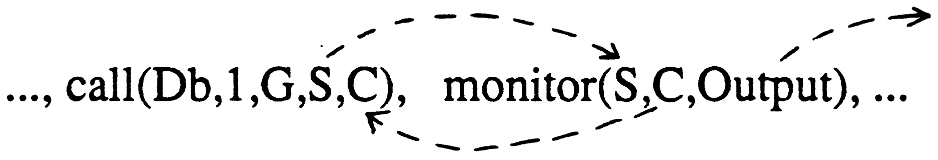
(3) The monitoring program can instantiate *NG* to a new goal (in the example, false)

Applications:

- implement alternative exception handlers (including inheritance, query-the-user, ...)
- enhance expressive power of language by implementing system calls

A Simple Exception Handler

- Reports termination
- Aborts program if any run-time errors reported
- Closed world: fails all goals unsolvable in program


..., call(Db,1,G,S,C), monitor(S,C,Output), ...

```
mode initiate(Db?,G?,Output↑).

initiate(Db,G,O) ← call(Db,1,G,S,C), monitor(S,C,O).


mode monitor(Status?,Control↑,Output↑).

monitor(failed(_),C,[failed]).

monitor(succeeded,C,[succeeded]).

monitor([exception(T)|_],stop,[exception(T)]).

monitor([exception(T,G,NG) | S],C,[exception(T,G) | O]) ←
        NG = false, monitor(S, C, O).
```

Concurrency Control in PPS

- PPS is a multiprogramming system: several queries can execute concurrently

- PPS supports metarelations that permit a program to:

  - access terms representing system state (*definition*, ...)
  - generate new states (*new_definition*, ...)
  - specify a new state, to apply on success

  State is the set of all databases defined in PPS

- A PPS program may thus access a database in three ways:

  - to execute its code
  - to read its source
  - to assert new versions of predicates

- Control mechanisms are required to:

  - avoid contention due to concurrent access/update
  - maintain declarative semantics: a PPS program is a relation over states

- Implementation of these mechanisms illustrates use of PARLOG for systems programming

Metarelations

```
current(State↑)
definition(State?,Db?,Relation?,Definition↑)
new_definition(State?,Db?,Definition?,NewState↑)
next(State?)
etc ...
```

1. Program transformation

Apply a transformation to all predicates in *Db*.

```
transform_db(Db) ←
    current(S),
    dict(S,Db,Dict),
    transform_db(S,Db,Dict,S'),
    next(S').

mode transform_db(State?,Db?,Dict?,NewState↑).
transform_db(S,Db,[R/Dict],S'') ←
    definition(S,Db,R,Defn),
    transform(Defn,Defn'),
    new_definition(S,Db,Defn',S'),
    transform_db(S',Db,Dict,S'').
transform_db(S,Db,[ ],S).
```

2. Alternative Worlds

Simulate execution of *Q* in *Db* and in {*Db* + *Fact*}

```
try(Db,Q,Fact) ←
    current(S),
    add_fact(S,Db,Fact,S'),
    demo(S,Q),
    demo(S',Q).
```

Concurrency Control: Implementation

- For each query, PPS must:
  - maintain alternative states
  - commit modifications on successful termination of query
  - abort committment if conflicting accesses
- Queries and databases are represented as goals in a parallel conjunction: that is, as processes.
- Queries communicate with databases by query (*q*) and metaquery (*mq*) messages:

  message(To,q(Q,Done),Result)
  message(To,mq(Q,Done),Result)

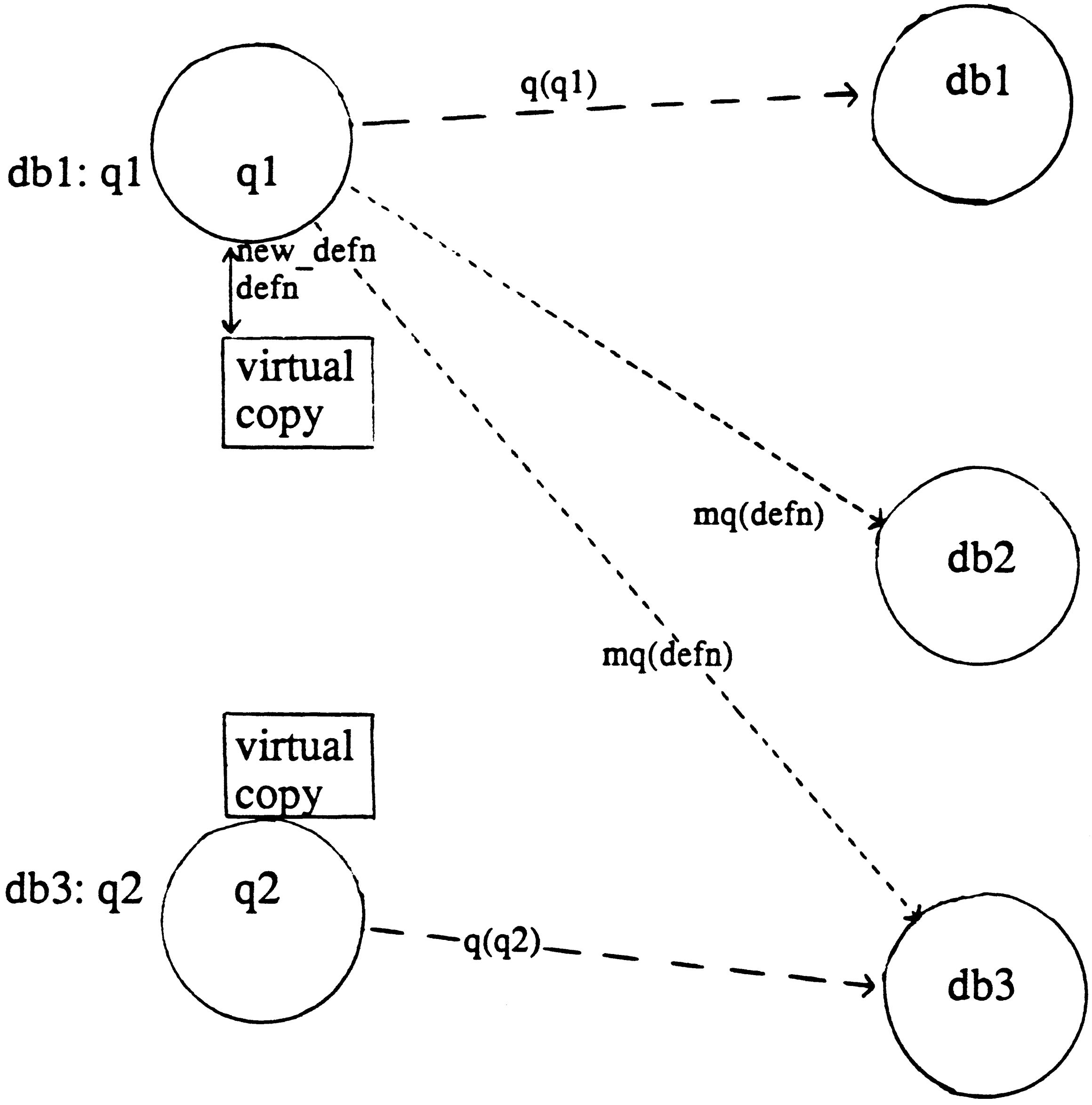
- Incomplete messages are used to return results of requests

  Result = true, Result = error(Q)

- A *Done* variable associated with a query is included in all messages. This is bound when query terminates. This is an example of a synchronization variable.
- Updates (new states) are cached in the query process and applied to databases on success using a two-stage commit procedure

Concurrency Control: Implementation

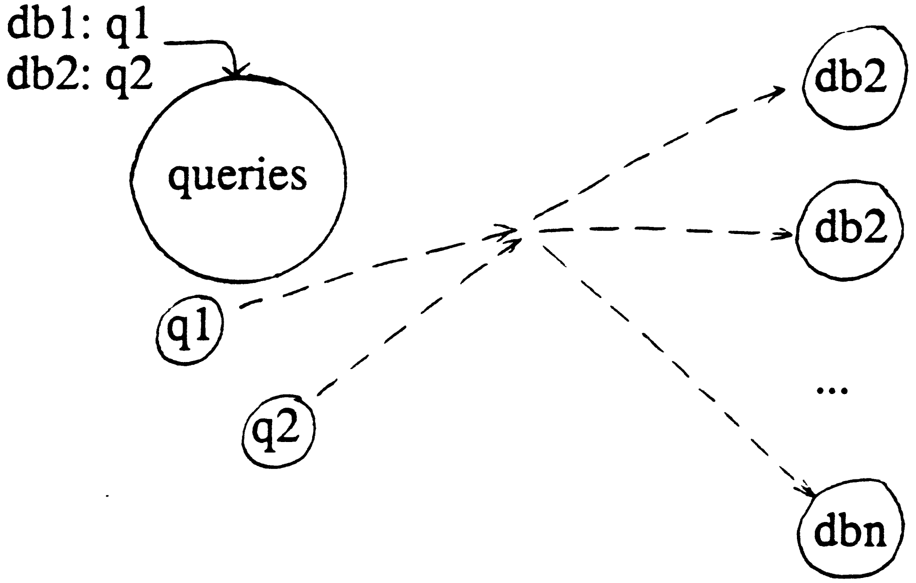
Assume three databases, db1, db2, db3.
Two concurrent queries, db1: q1 and db3: q2



- Virtual copies record modifications to databases
- PPS applies updates if query succeeds
- Updates are not applied if concurrent queries active in any database modified by query

PPS Structure

init(Qs) ← queries(Qs,DbRs), databases(DbRs).



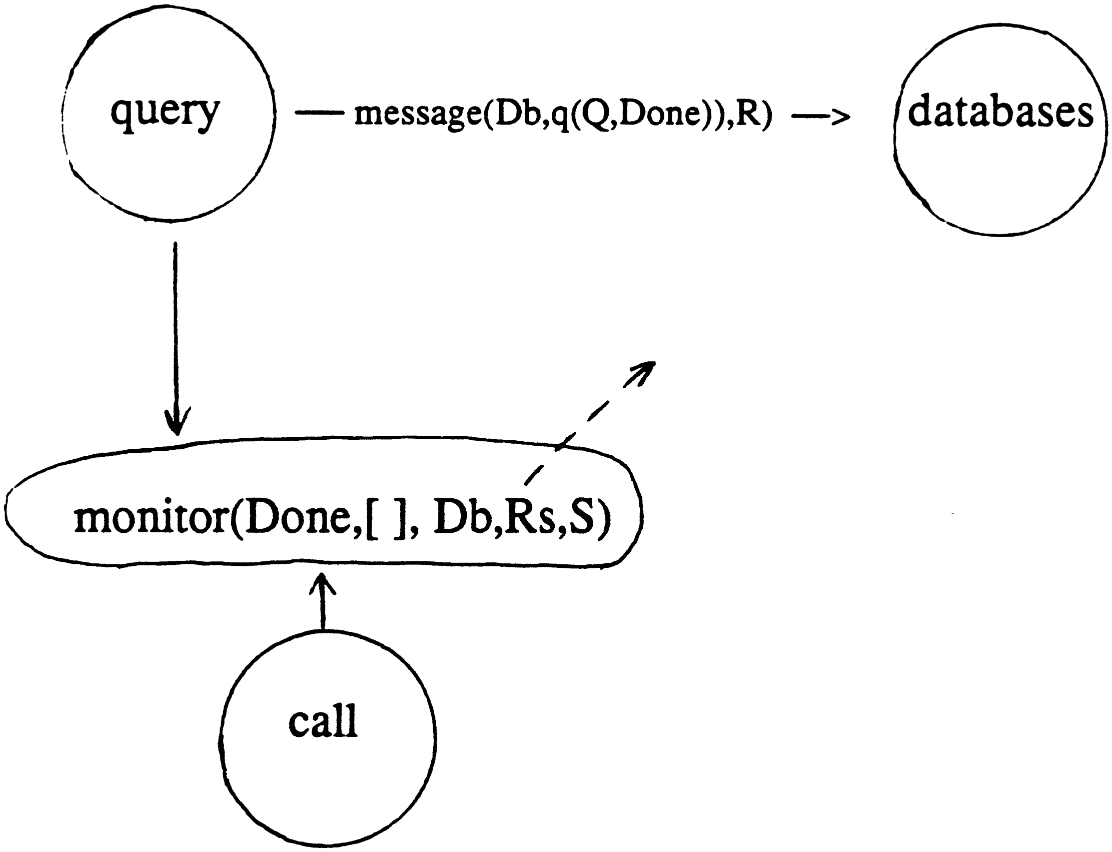
```
mode queries(Queries?, DbRequests↑).
queries([ (Db : Q) | Queries], DbRs) ←
        query(Db, Q, QRs),
        merge(QRs, Rs, DbRs),
        queries(Queries, Rs).
```

- *queries* process waits for query messages
- Spawns a *query* process to control query evaluation
- Merges *query*'s database request stream into general stream

The Query Process

- Generates a query request message to the specified database
- Starts computation using the metacall
- Spawns a monitor process to control evaluation

```
mode query(Db?,Query?,DbRequests↑).

query(Db,Q,[message(Db,q(Q,Done),R) | Rs]) ←

        monitor(Done,[ ],Db,Rs,S),

        call(Db,1,R,S,C).
```



The Monitor Process

- The monitor process controls a query evaluation
- It responds to query status messages by accessing *States* and, if necessary, generating messages to databases

```
mode monitor(Done?,States?,Db?,Requests↑,Status?).
```

Failure:

```
monitor(D,States,Db,[ ],failed(_))← D = failed.
```

Success:

```
monitor(D,States,Db,Requests, succeeded) ←
        commit(   States,Requests) & D=done.
```

Metarelations: (definition and new_definition)

```
monitor(D,States,Db, Rs,[ exception(meta, G, R) / S]) ←
    meta_relation(D,States,Rs,G,R,States',Rs'),
    monitor(D,States',Db,Rs', S).
```

Calls to relations in other databases:

```
monitor(D,States,Db,[message(ODb,q(G,D),R) / Rs],
        [ exception(undef,ODb#G,NG) / S]) ←
    NG = call(ODb, R ),
    monitor(D,States,Db,Rs,S).
```

On Success, Two-Stage Commit

(1) Generate a commit message to each database to be updated

commit(Updates,Oki,Go), Oki and *Go* variables

(2) If all *Oki = ok*, instantiate *Go* to *go*

If any *Oki = abort*, instantiate *Go* to *abort*

```
mode commit(States?, Requests↑).
commit(States,Requests) ←
        updates_required(States,Updates) ,
        request_commit(Updates,Requests,Go,Oks),
        confirm_commit(Oks,Go).

mode request_commit(Updates?, Rs↑,Go↑, Oks↑).
request_commit([db(Db,Us) | Updates],
          [ message(Db,commit(Us,Ok, Go),R) | Rs ], Go,
          [Ok | Oks]) ←
        request_commit(Updates,Rs,Go,Oks).
request_commit([ ], [ ], Go, [ ]).

mode confirm-commit(?,↑).
confirm_commit([ok | Oks],Go) ←
        confirm_commit(Oks,Go).
confirm_commit([abort | Oks], abort).
confirm_commit([ ], go).
```

Database Processes

- Each PPS database is represented by a database process
- Each has a stream connection to a disk process, which it can request to retrieve and store source and object code
- A database accepts queries, metaqueries and commits.
- Queries and metaqueries are processed at any time, and their *Done* variable recorded
- Commits are only processed if there are no concurrent accesses to database (all *Done* variables are bound).

```
mode database(Rs?,Db?,Source?,DoneVars?).

database([ message(Db,q(Q,D),R) | Rs],Db,S,DVs) ←

        R = Q, database(Rs,Db,S,[D | DVs]).

database([ message(Db,mq(Q,D),R) | Rs],Db,S,DVs) ←

        handle_mq(Q,S,    R),

        database(Rs,Db,S ,[D | DVs]).

database([ message(Db,commit(Us,Ok,Go),R) | Rs]
                                              Db,S,DVs) ←
        try_commit(Us,Ok,Go,DVs,DVs',S,S') &

        database(Rs,Db,S',DVs').
```

Commit ment: Databases

- A *commit(Us,Ok,Go)* message indicates a query wishes to commit
- If no concurrent executes or reads to database, signal that commit may proceed (*Ok = ok*)
- If query signals that commit should proceed (*Go = go*) apply updates *Us*

```
mode try_commits(Us?,Ok↑,Go?,DVs?,DVs'?,S?,S'↑).
try_commits(Us,ok,Go,DVs,[ ],S,S') ←
        check(DVs),
        valid_updates(Us) :
        try_commits2(Us,Go,S,S');          % Note sequential OR
try_commits(Us,abort,Go,DVs, DVs,S,S).


mode try_commits2(Updates?,Go?,Source?,Source'↑).
try_commits2(Us,go,S,S') ← apply_updates(Us,S,S').
try_commits2(Us,abort,S,S).


check([D | DVs]) ← not(var(D) ) : check(DVs).
check([ ]).
```

PPS: Summary

- The PPS is a sophisticated programming environment / operating system for PARLOG, written in PARLOG
- Individual relations in its implementation can be read as specifications of programming environment components
- Components execute concurently
- Directional unification (dataflow) is used to constrain reduction to obtain desired operational behaviour
- Back communication and synchronization variables provide a succinct representation of a parallel algorithm: distributed database update
- The control metacall is used to initiate, monitor and control user computation
- Exception messages provide for communication between user tasks and operating system

PARLOG Implementation

Two approachs: *Shallow or-parallel* (*non-flat*) and *flat.*

- *Non-flat*: user defined predicates in guards ⇒ *AND-OR tree.*
- *Flat*: only simple tests in guards ⇒ *process pool.*

Non-flat implementation work: restricted to PARLOG.

Flat implementation work: Flat PARLOG; FCP; FGHC.

(Flat PARLOG + metacall = Full PARLOG)

- **Non-flat PARLOG**: *AND-OR tree* computational model

Sequential PARLOG Machine (SPM)	Sequential
PARLOG on Prolog	Sequential
**Shared Memory Multiprocessors	Parallel

- **Flat PARLOG**: *Process pool* computational model

ALICE packet rewrite machine	Parallel
**Flat PARLOG Machine (FPM)	Sequential/ (Distributed)

- **PARLOG control metacall**

Sequental PARLOG Machine
**Flat PARLOG Machine

*(** = described here)*

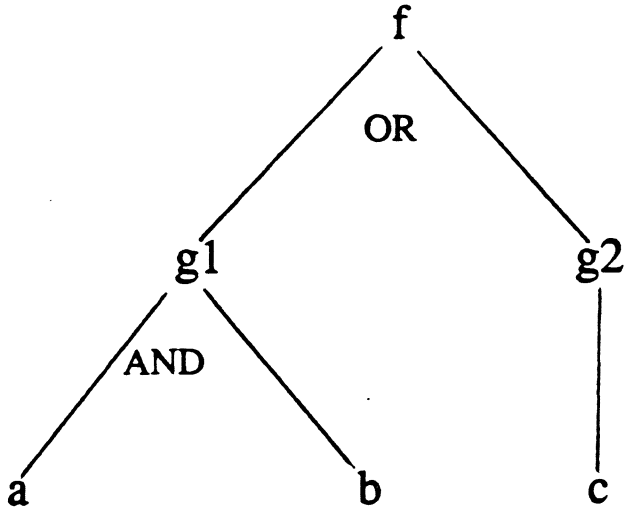
Shallow Or-Parallelism in PARLOG

Deep guards mean that program execution may create an *AND-OR tree.*

PARLOG's *guard safety* mean that multiple environments are *not* required during OR-parallel evaluation

$f \leftarrow g1 : b1.$ $\quad g1 \leftarrow a, b.$ $\quad g2 \leftarrow c.$

$f \leftarrow g2 : b2.$



AND/OR Tree

Nodes may be regarded as processes

Non-leaf nodes await evaluation of offspring

Leaf nodes correspond to reducable or suspended processes

Crammond's Process Model

Two types of process:

Goal processes - (AND processes)
responsible for creating child clause processes

Clause processes - (OR processes)
responsible for executing a single clause

Process States:

Runnable, executing	Suspended on variable
Runnable, queued	Suspended on child

Execution:

- A goal process creates a clause process for each clause
- Each clause process spawns goal processes to solve guard goals
- First clause process to succeed with guard commits and spawns goal processes to solve body goals
- Processes communicate by signals

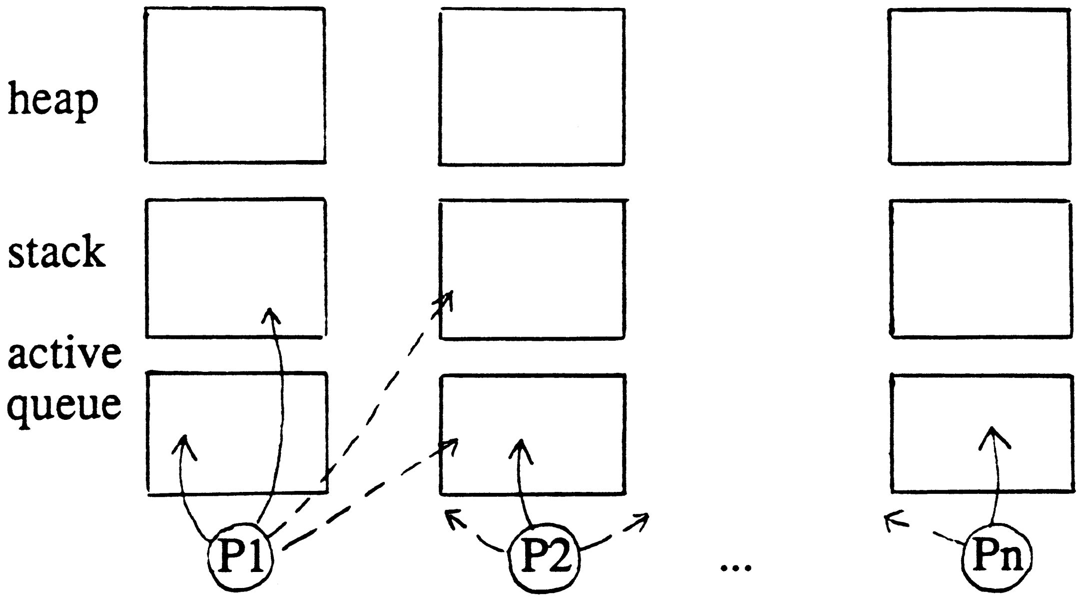
Process Signals: **DONE + QUIT**

Goal process succeeds ————> **DONE**
Goal process fails ————————> **QUIT**

Clause process succeeds ————> **QUIT**
Clause process fails ————————> **DONE**

Implementation: Sequent Multiprocessor

- Sequent: shared memory multiprocessor with hardware locks
- One Unix process / processor; executes PARLOG processes
- Each processor has its own data areas
- Locks used when writing shared data areas
- Procedures are encoded using Warren-like code
- Processes are represented as data structures



- Processors 'steal' processes from neighbours when necessary
- Optimisations for flat programs:

  do not generate clause processes if a test on a single argument can reduce choice to a single clause

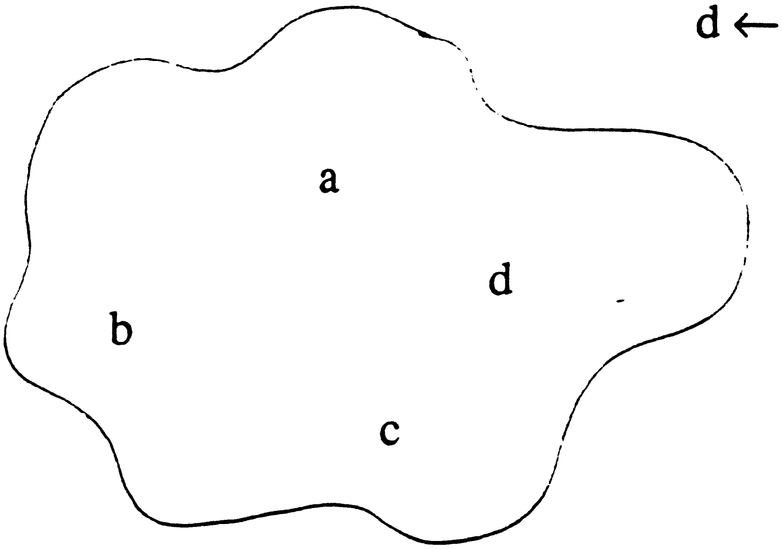
- Significant speedups achieved: 4.6 times on 5 processors

Flat PARLOG Computational Model

- Computation = process pool

p ← a,b,c,d.

d ← g1 : b1,u,v.
d ← g2 : b2.



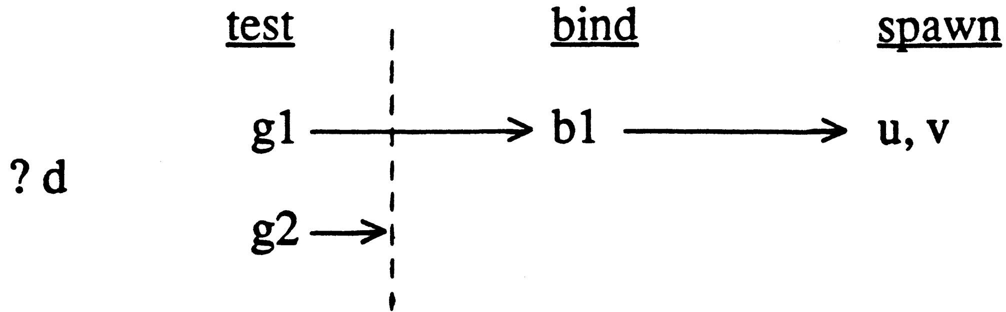
- Repeatedly select and attempt to reduce processes using clauses defining procedure

- Three phase reduction cycle:

test then **bind** then **spawn.**

Input mode arguments + guard calls define *tests*

Output mode arguments + '=' calls define *binds*



test | bind | spawn

? d

g1 → b1 → u, v

g2 →

- A process try may <u>succeed</u>, <u>fail</u> or <u>suspend</u>.

Compilation of Flat Parlog

Flat Parlog:

mode lookup(Key?, Data^, Dict?, NewDict^).

lookup(K, *D*, [{K,D} | Ds], *[{K,D} | Ds]*) .

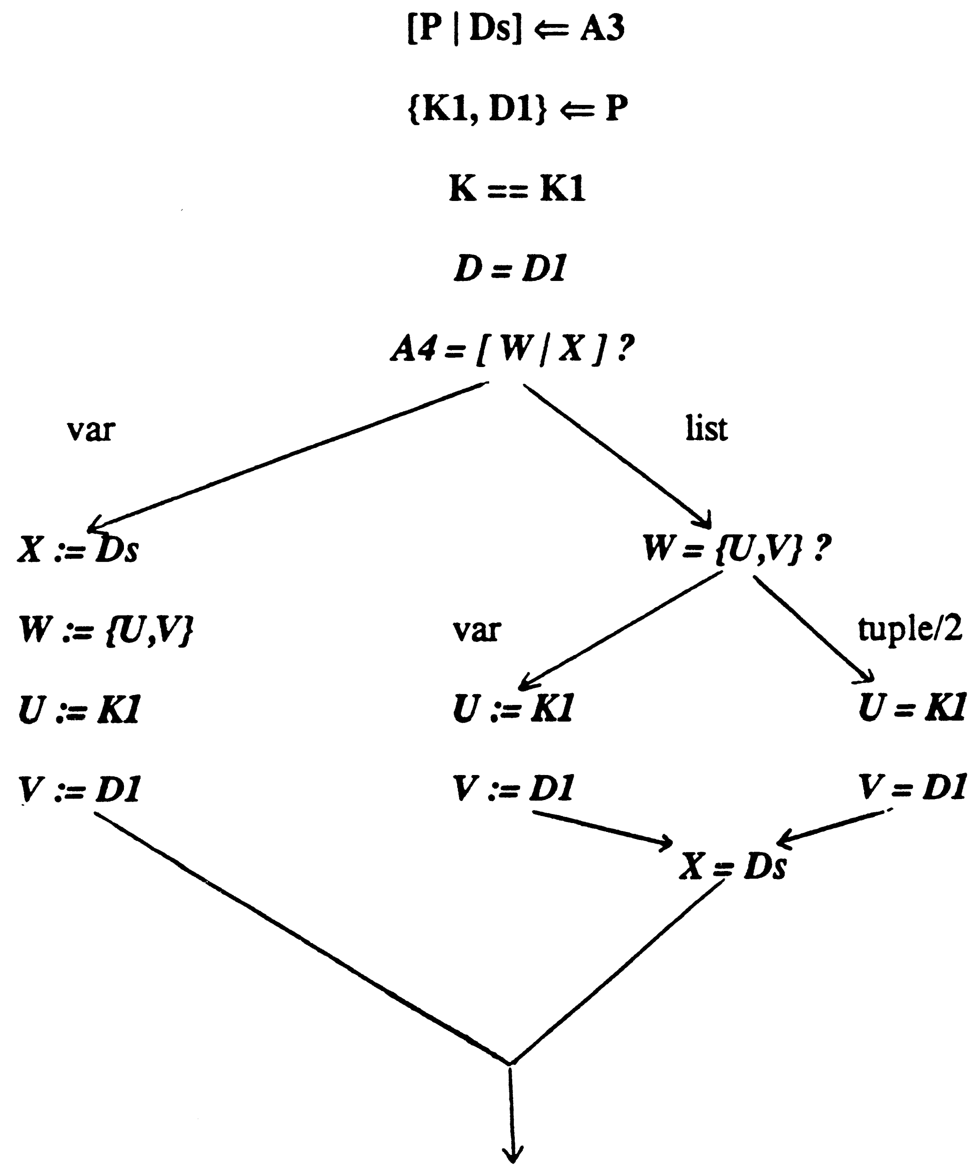
Standard Form:

lookup(K,D, A3,A4) ←

[{K1,D1} | Ds] ⇐ A3, K == K1 : *D = D1, A4 = [{K1,D1} | Ds].*

<test> ***<bind>*** **(<spawn>)**

Compiled Form:



[P | Ds] ⇐ A3

{K1, D1} ⇐ P

K == K1

D = D1

A4 = [W / X] ?

var

X := Ds

W := {U,V}

U := K1

V := D1

list

W = {U,V} ?

var

U := K1

V := D1

tuple/2

U = K1

V = D1

X = Ds

Compilation of Flat Parlog *contd*

- Compile unification to basic operations:

T ⇐ V	Input matching	*test_integer, test_list*
V := T	Assignment	*put_integer*
V = T	Binding	*bind_integer, bind_list*
V = V	General unification	*unify*
V == V	Equality	*equals*

- All instructions single mode (alternative: moded instructions)
- Input matching *and* output unification compiled

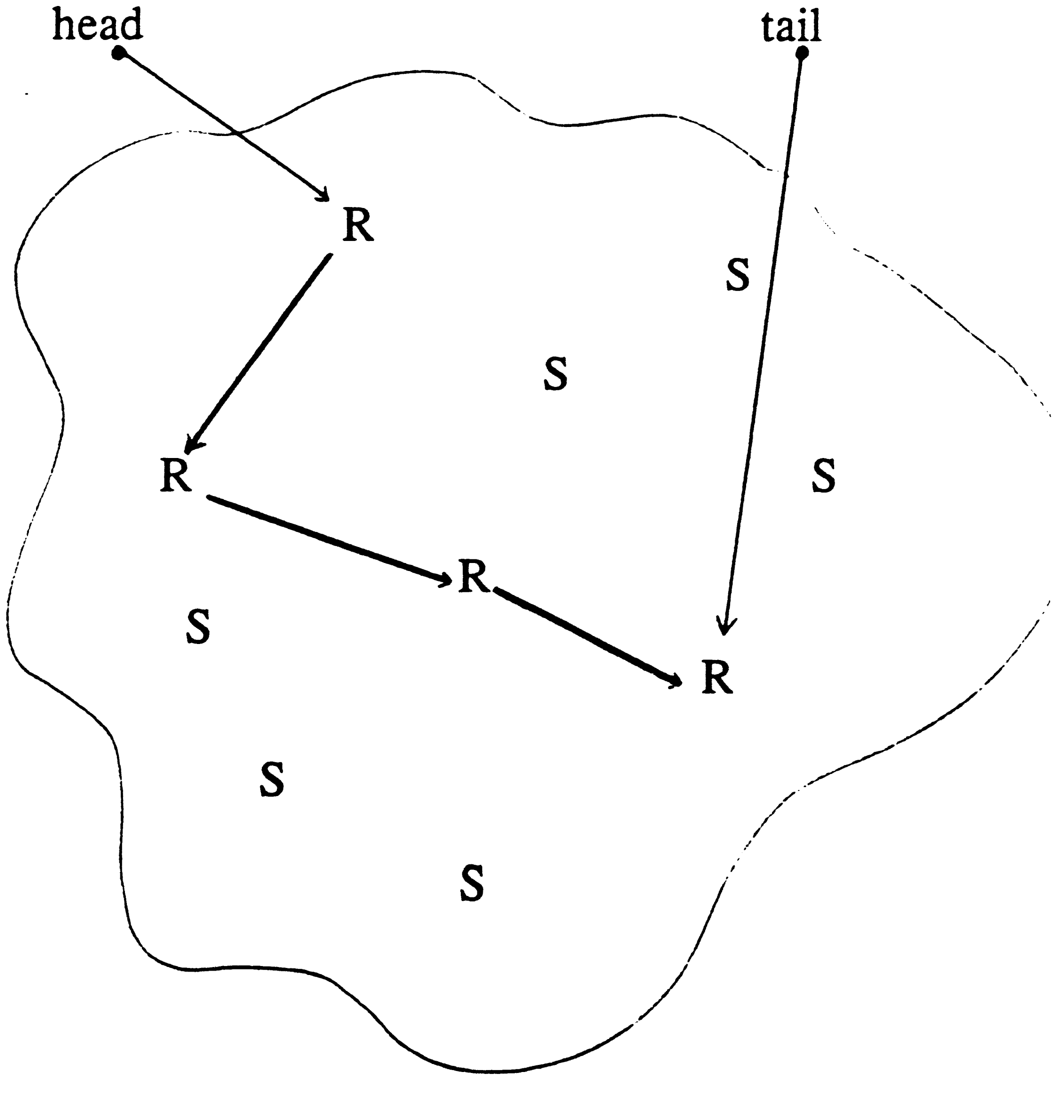
```
Lookup/4:
   load(4)
   try_me_else(Lookup4)

         test_list(2,4)              [P | D] ⇐ A3
         test_tuple(4,2,6)           {K1,D1} ⇐ P         <test>
         equal(6,0)                  K == K1

         unify(1,7)
         bind_list(3)
         get(8)
         put_value(5)
         put_tuple(2,8)                                  <bind>
         put_value(0)
         put_value(7)
         halt
      Lookup1:
         get(2,8,11)
         bind_tuple(8,2,Lookup3)
         put_value(0)
         put_value(7)
      Lookup2:
         unify(11,5)
         halt
      Lookup3:
         get(2,9,10)
         unify(9,0)
         unify(10,7)
         goto(Lookup2)
Lookup4:
```

Note absence of a commit operator

Flat Parlog Machine

Process pool computational model +

- scheduling structure:
  to avoid selecting suspended processes
- tail recursion:
  to avoid repeated selection of processes



PARLOG Control Metacall

call(Module,Priority,Goal,Status,Control)

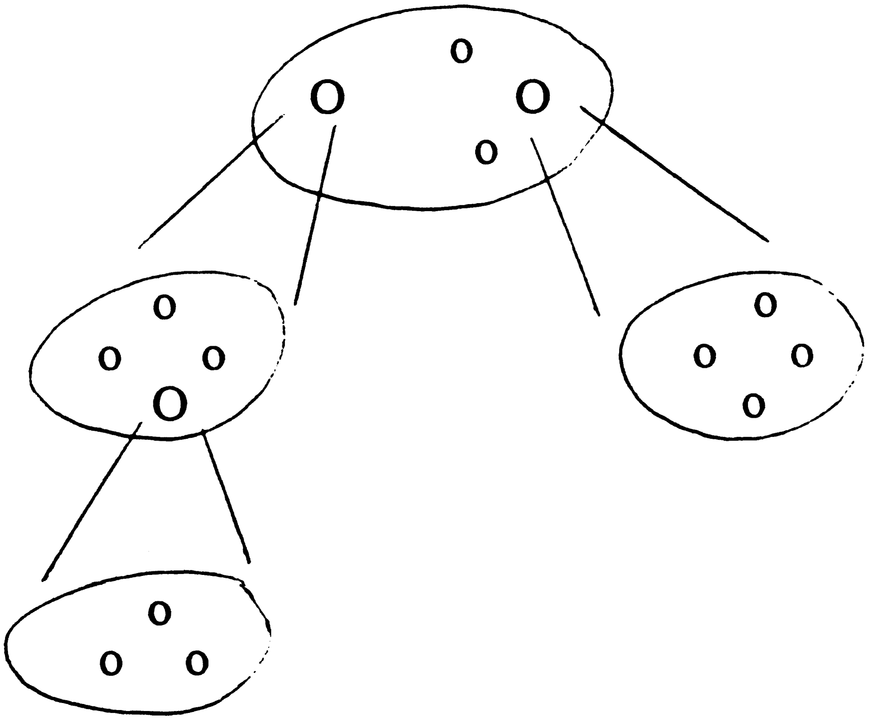
- Initiate a computation
- Signal termination of a computation
- Allow control of a computation
- Provide fair and prioritized scheduling of a computation

Implementation via extensions to Flat PARLOG Machine

(a) New computational model

Flat PARLOG + metacall = computation tree

A process pool may contain subpools



- A subpool is a computation or task
- Reduction repeatedly selects {computation,process} pair

Extensions to Flat PARLOG Machine, contd:

(b) Allow for termination of a computation

- process failure = success of computation + failed status message
- no processes in a computation = success of computation
- no reducable processes in a computation = computation deadlock

(c) Fair and prioritized computation scheduling

- fairness = bounded depth-first round-robin scheduling
- priorities = directed selection of {computation, process} pair

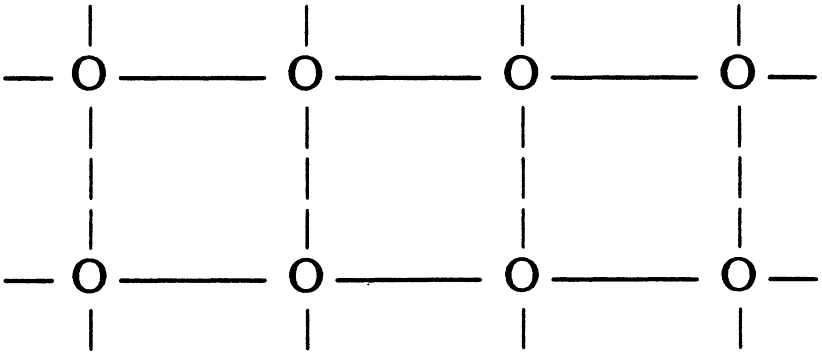
Benchmarks on uniprocessors (SUN-3) show only 2-3 % performance degradation when FPM is extended to support control metacall.

Parallel Execution of Flat PARLOG

(Based on Taylor's FCP Implementation)

Assume:

- many processors connected by regular interconnection topology (mesh, hypercube)
- no global memory; each processor has local memory
- processors may communicate by message passing
- Flat PARLOG computation (process pool) distributed over many processors
- each processor executes Flat PARLOG Machine



Three issues:

- distributed unification (discussed here)
- distributed computation control (metacall)
- load balancing, code mapping

Distributed Unification

PARLOG:

```
mode f(?,^).
f(1,2) <- g.
```

Standard form:

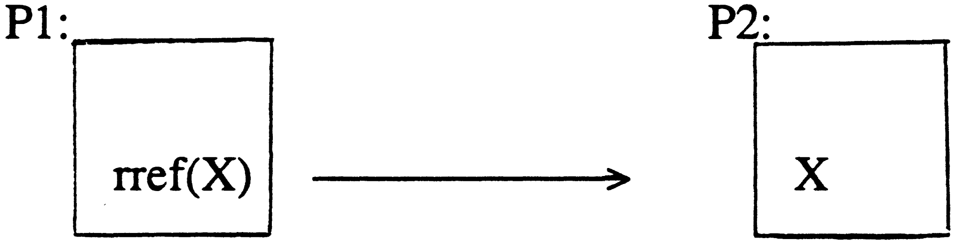
$$f(X0,X1) \leftarrow X0 \Leftarrow 1 \; : \; X2 = 2, g.$$

Reduction Cycle:

test: $X0 \Leftarrow 1$? (read) } may encounter
} remote references
bind: $X1 = 2$ (write) }

spawn: g

Remote reference = processor number + address.



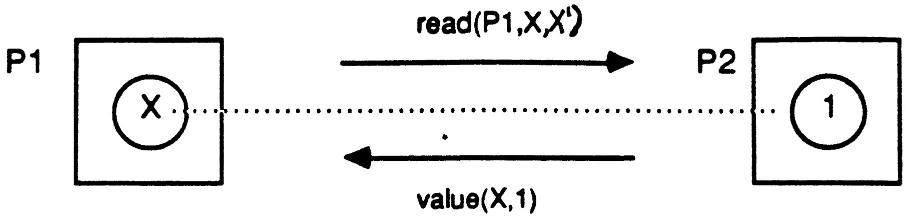
Unification + remote refs = extended unification algorithm:

- test phase: generate read messages and suspend process if remote references encountered (e.g. *X0* is remote)
- bind phase: generate unify message if remote reference encountered (e.g. *X1* is remote)

Messages dereference remote references until values or variables found

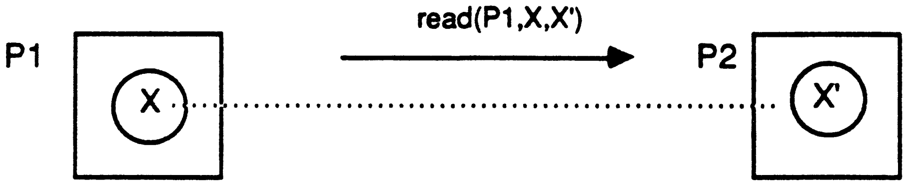
Distributed Unification: Reading

1. X <= 1 : value is available.

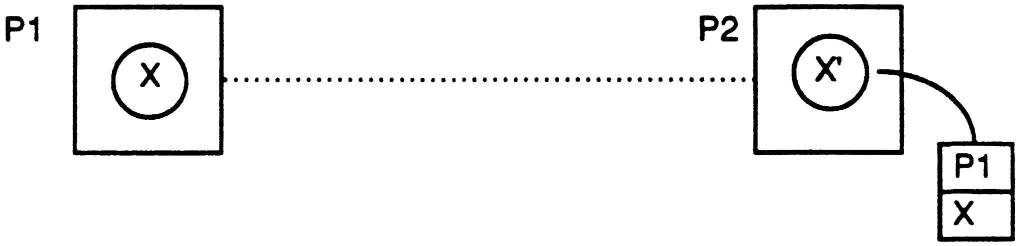


2. X <= 1 : value is not available

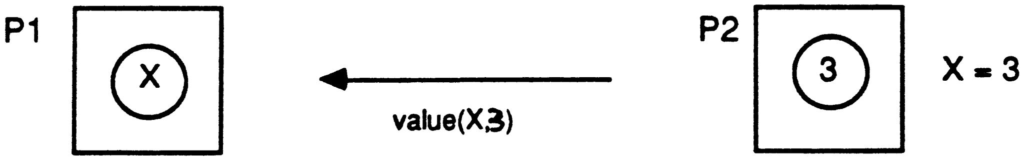
(a)



(b)



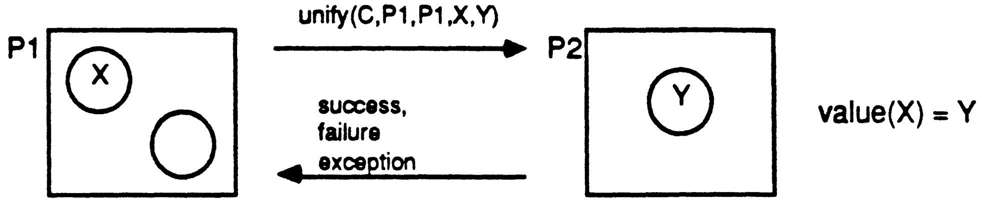
(c)



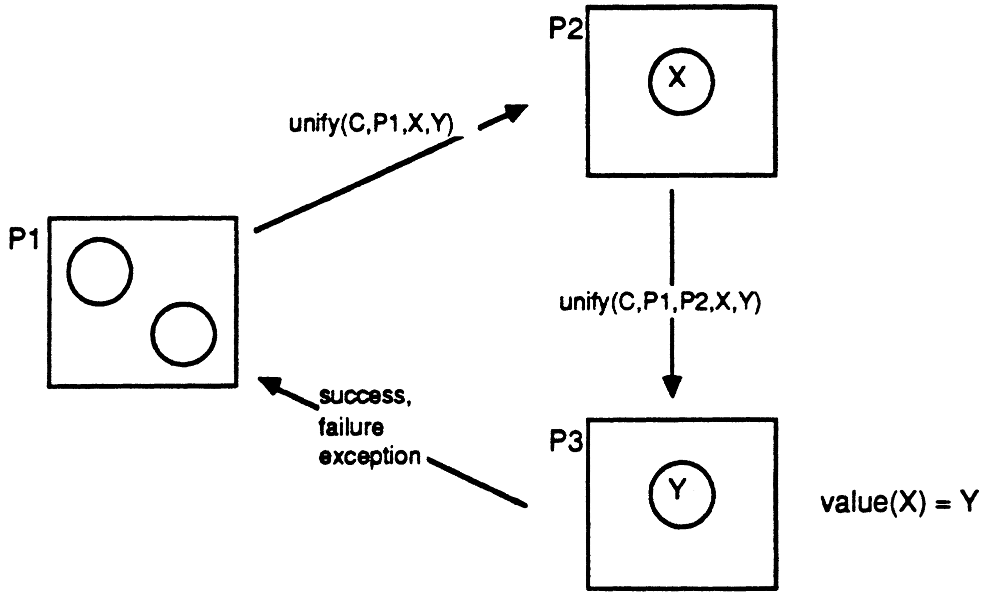
Distributed Unification: Writing

(a) Both local

(b) One local,
One remote



(c) Both remote



Recent Developments

1. Move to Flat Languages

Recall: A flat parallel logic language only permits calls to system primitive predicates in clause guards.

Why flat?

(a) Necessity

CP	—>	FCP	) appears
			) essential for
GHC	—>	FGHC	) implementation

(b) Convenience

PARLOG —> Flat PARLOG simplicity + efficiency

(Recall: Flat PARLOG + metacall = PARLOG)

Justification: two suppositions:

"Most programs are flat"

"Flat languages can be implemented more efficiently"

Both of which are true ... some of the time

- certain applications (e.g. AI) make extensive use of PARLOG's shallow OR-parallelism
- clever PARLOG implementations (e.g. Crammond's) can approach flat language efficiency

Recent Developments

2. Restoring Completeness

PARLOG and GHC are correct but incomplete logic languages

In certain applications, this lack of completeness is a disadvantage

2.1. P-Prolog

- incorporates exclusive relations
- synchronization mechanism: an exclusive relation cannot reduce until at most one clause is applicable

  Example:

  append([X / Xs], Ys, [X / Zs]) ← append(Xs,Ys,Zs).
  append([ ], Ys, Ys).

  This cannot reduce until its first OR second and third arguments are instantiated.

- Permits multi-moded relations
- Alternative syntax permits non-exclusive relations
- Not clear that P-Prolog can be efficiently implemented

Recent Developments

2.2 PARLOG & Prolog United

- PARLOG has two set constructor interfaces to pure Horn clauses:

  set(Solutions↑, Term?, Conjunction?)
  incrementally binds *Solutions* to a list of solutions

  subset(Solutions?,Term?,Conjunction?)
  generates solutions as *Solutions* is bound to a list of variables

- Recently, Clark and Gregory propose linking PARLOG and Prolog more tightly
- General two-way communication between Prolog and PARLOG computations
- Applications and implementation not yet clear

2.3 Compiling Prolog to Parallel Logic Languages

- Ueda, Tamaki describe techniques for compiling Horn clause programs to GHC
- Can handle exhaustive search programs and generate and test
- Restrictive: modes must be specified at compile-time

Who's Using Parallel Logic Languages?

(Partial list)

Research groups:

PARLOG Group, Imperial College, London.
ICOT Research Centre, Tokyo.
Weizmann Institute, Israel
Vulcan Group, Xerox PARC

Industry in Uk, Sweden, Japan, ...

Implementations

Distribution

PARLOG Group: PARLOG on Unix and Prolog
Weizmann: FCP on Unix
ICOT: FGHC on Prolog

Experimental

PARLOG Group: Sequent, ALICE, Flat Parlog Machine
Weizmann: Hypercube
ICOT: multi-PSI
others in UK, Sweden, Japan, ...

Directions for Future Development

- Fast parallel implementations

  On custom hardware:
    ICOT: multi-PSI 100K RPS/PSI-II x 100 = ...

  Conventional machines
    native code compilation of FCP: 75K RPS on SUN-3s

- Formal semantics

  facilitating:
    program transformation
    program analysis and debugging

- Declarative programming environments

- Language extensions and new languages

  restoring completeness
  constraints
  typing

- New applications

  distributed systems
  AI

- Commercial exploitation

  as a specification and implementation language for distributed systems

  for parallel symbolic processing

Conclusions

- Parallel logic languages =

Horn clause logic + concurrent evaluation
dataflow synchronization
committed-choice non-determinism

- Declarative content makes parallel logic programs:

easy to understand
easy to transform and analyze

- Parallel evaluation permits parallel symbolic processing
- Operational characteristics make parallel logic languages a powerful programming formalism
- Efficient parallel implementations are being developed
- Broad range of applications
- Parallel logic languages are a viable language for expressing and implementing parallel algorithms

Selected References

1. Languages

Clark, K.L., and Gregory, S. 1981, "A relational language for parallel programming". In *Proc. 1981 ACM Conf. on Functional Programming Languages and Computer Architectures* (Portsmouth, NH), pp. 171-178. The original paper.

Clark, K.L., and Gregory, S. 1986, "PARLOG: parallel programming in logic". In *ACM Trans. on Programming Languages and Systems*, 8 (1), pp. 1-49.

Clark, K.L., and Gregory, S. 1987, "PARLOG and Prolog united". In *Proceedings of the 4th International Logic Programming Conference*, (Melbourne, May), J.-L. Lassez (Ed), MIT Press.

Foster, I.T. and Taylor, S. 1987, "Flat PARLOG: a basis for comparison". Research report DOC 87/5, Imperial College, London. Contrasts Flat PARLOG and FCP, presents benchmarks comparing their efficiency and describes an abstract machine for the implementation of Flat PARLOG.

Gregory, S. 1987 , *Parallel Logic Programming in PARLOG*. Reading, Mass.: Addison-Wesley. The language, its applications and its implementation.

Mierowsky,C., Taylor, S., Shapiro, E., Levy, J., and Safra, M. 1985, "The design and implementation of Flat Concurrent Prolog". Technical Report CS85-09, Weizmann Institute, Rehovot, 1985.

Shapiro, E.Y. 1987, "Concurrent Prolog: a progress report". In IEEE Computer. Reviews the current status of research on Concurrent Prolog and its applications.

Ueda, K. 1986, *Guarded Horn Clauses*, EngD thesis, University of Tokyo. To be published by MIT press.

Ueda, K. 1987. "Making exhaustive search programs deterministic - part II". In *Proceedings of the 4th International Logic Programming Conference*, (Melbourne, May), J.-L. Lassez (Ed), MIT Press.

Yang, R. and Aiso, H. 1986, "P-Prolog: a parallel logic language based on exclusive relation". In *Proc. of the 3rd Intl. Logic Programming Conf.* (London, July), E.Shapiro (Ed.), NewYork: Springer-Verlag, pp 255-269.

2. Applications

Armstrong, J.L., Elshiewy, N.A. and Virding, R. 1986, "The phoning philosophers problem". In *Proc. of 1986 IEEE Symp. on Logic Programming* (Salt Lake City, Utah), pp 28-33. Describes the use of PARLOG to control telephone exchanges.

Clark, K.L., and Foster, I.T. 1987, "A declarative environment for concurrent logic programming". In *Proc. TAPSOFT '87* (Pisa, March). Describes PPS.

Foster, I.T. 1987, "Logic operating systems: design issues". In *Proceedings of the 4th International Logic Programming Conference*, (Melbourne, May), J.-L. Lassez (Ed), MIT Press. Describes extended PARLOG metacall and its application.

Gregory, S., Neely, R. and Ringwood, G.A. 1985, "PARLOG for specification, verification and simulation". In *Proc. of the 7th Intl Symp. on Computer Hardware Decription Languages and their Applications* (Tokyo, August), C.J. Koomen and T. Moto-oka (Eds), Amsterdam: Elsevier/North Holland, pp 139-148.

Taylor, S., Av-Ron, R. and Shapiro, E.Y. 1986. "A layered method for process and code mapping". *Journal of New Generation Computing*. Describes the use of a parallel logic language to describe load balancing and code mapping on a distributed machine.

Shapiro, E.Y. and Takeuchi, A. 1983. "Object-oriented programming in Concurrent Prolog". In *Journal of New Generation Computing*, 1(1).

3. Implementations

Crammond, J. 1986. "An execution model for committed-choice non-deterministic languages". In *Proc. of 1986 IEEE Symp. on Logic Programming* (Salt Lake City, Utah), pp 148-158.

Foster, I.T., Gregory, S., Ringwood, G. A., and Satoh, K. 1986 , "A sequential implementation of PARLOG". In *Proc. of the 3rd Intl. Logic Programming Conf.* (London, July), E.Shapiro (Ed.), NewYork: Springer-Verlag, pp 149-156.

Ichiyoshi, N., Miyazaki, T. and Taki, K. 1987. "A distributed implementation of Flat GHC on the Multi-PSI". In *Proceedings of the 4th International Logic Programming Conference*, (Melbourne, May), J.-L. Lassez (Ed), MIT Press.

Lam, M. and Gregory, S. 1986 , "PARLOG on ALICE: a marriage of convenience". Research report, Department of Computing, Imperial College, London. In *Proceedings of the 4th International Logic Programming Conference*, (Melbourne, May), J.-L. Lassez (Ed), MIT Press.

Taylor, S., Safra, S. and Shapiro, E.Y. 1987 , "A distributed implementation of Flat Concurrent Prolog". In *International Journal of Parallel Processing* 15(3), pp 245-275. Excellent description of a Hypercube implementation of FCP.