IMPLEMENTATION OF ANDORRA-BASED LANGUAGES

(ANDORRA-BASED LANGUAGES AND THEIR IMPLEMENTATION)

Sverker Janson

Swedish Institute of Computer Science
Box 1263, S-164 28 Kista, Sweden
tel +46-08-752 15 72, fax +46-08-751 72 30
e-mail sverker@sics.se

ICLP tutorial, June 13, 1994

This tutorial

ANDORRA “CLASSIC”

- Basic Andorra Model (BAM)
- Advantages of the BAM
- Problems with Prolog+BAM
- Implementation

CCP LANGUAGES

- Flat CCP
- Deep CCP
- Stability
- HO CCP

IMPLEMENTATION OF DEEP CCP

- Basic issues
- Establishing stability
- Copying vs sharing
- Avoiding copying

Basic Andorra Model (BAM)

D. H. D. Warren (in 1987) proposed to use the notion of determinism to exploit dependent and-parallelism in logic programs and coined the name Andorra
(cf., Rong Yang's P-Prolog).

- A goal is *determinate* if it has at most one candidate clause.
- Determinate goals are executed first.
- When no goal is determinate, a goal is selected for nondeterminate execution.

Advantages of the BAM

- Programs are less execution order sensitive than in Prolog.
- It is possible to model producer-consumer relationships, and thereby control concurrency.
- All goals that are determinate can be executed in parallel.
- The lazy style of nondeterministic execution has a good effect on search programs.

Less sensitive to execution order

Infinite number of backtrack points in Prolog:

```
?- member(X, L), L = [1,2,3].
```

Finite:

```
?- L = [1,2,3], member(X, L).
```

In BAM both are finite.

Supports concurrent programming

Determinism synchronises producers and consumers.

```
n(0, []).
n(N, [N|R]) :- N > 0, N1 is N-1, n(N1, R).

s([], 0).
s([N|R], S) :- s(R, S1), S is N+S1.

?- n(42, L), s(L, S).
```

Extracts implicit parallelism

```
qs([], S, S).
qs([C|R], S0, S) :- p(R, C, Le, Gt),
      qs(Le, S0, [C|S1]), qs(Gt, S1, S).

p([], _, [], []).
p([X|R], C, [X|Le], Gt) :- X ≤ C, p(R, C, Le, Gt).
p([X|R], C, Le, [X|Gt]) :- X > C, p(R, C, Le, Gt).
```

Goals with instantiated argument are determinate.
All determinate goals are reduced in one step.

```
?- qs([2,3,1]).

?- p([3,1]), qs(L1),                  qs(L2).

?- p([1]),   qs(L1),                  qs([3|L2']).

?- p([]),    qs([1|L1']),             p([]),  qs(L1''),  qs(L2'').

?-           p([]), qs(L1'''), qs(L2'''),     qs([]),    qs([]).

?-                  qs([]),    qs([]).

?-
```

Good effect on search

p(on, X, X).
p(off, _, _).

$A_2 \neq A_3 \rightarrow A_1 = \text{off}$

at_most_one([V_1, …, V_n]) :-
    p(V_1, 1, Y),
    …,
    p(V_n, n, Y).

$V_i = \text{on}$
$\rightarrow$
$Y = i$, $V_j = \text{off}$ $(j \neq i)$

q(on, _, _).
q(off, X, X).

$A_2 \neq A_3 \rightarrow A_1 = \text{on}$

at_least_one([V_1, …, V_n]) :-
    S_1 = a,
    q(V_1, S_1, S_2),
    …,
    q(V_n, S_n, S_{n+1}),
    S_{n+1} = b.

$V_j = \text{off}$ $(j \neq i)$
$\rightarrow$
$S_i = a$, $S_{i+1} = b$, $V_i = \text{on}$

Problems with Prolog+BAM

- Which goals are determinate?
- Concurrency breaks Prolog semantics
- Programs may do more work!

Which goals are determinate?

```
p(a, b).
p(a, a).

?- p(_, a).              – general indexing

?- p(X, X).              – more than indexing

q(0) :- ...              – more than head unification
q(N) :- N > 0, ...

r(0) :- !, ...           – pruning ...
r(N) :- ...
```

Concurrency breaks Prolog semantics

- Side effects may be permuted.

```
?- write(a), write(b).
```

- Side effects may be inhibited by early failure.

```
?- write(a), fail.
```

- Pruning may be inhibited by early failure.

```
?- p(X), !, q(X).

p(a).          q(b).
p(b).          q(c).
```

- Behaviour may be changed by early bindings.

```
?- var(X), X = a.

p(a) :- !.
p(_).

?- p(X), X = b.
```

Programs may do more work! (even loop)

```
p(a).
p(b).

q(c).
q(d).

?- p(X), q(X), lots_of_determinate_work.
```

Looping can be avoided by bounding the amount of determinate work.

Implementation

IMPLEMENTATIONS

- Andorra-I (Yang, Costa, Warren)

[• NUA-Prolog (Palmer, Naish)]

ISSUES

- Disallowing early execution
- Establishing determinacy of goals
- Maintaining order of goals
- Sharing of goal structure

[• Or-parallel environments]

[• Scheduling (both and and or)]

[• Memory management]

Andorra-I

Exploits transparently both dependent and-parallelism and or-parallelism in full Prolog.

Papers at ICLP89, ICLP91, PPoPP91, ICLP93.

Rong Yang, Inês Dutra, and Vítor Santos Costa. *Design of the Andorra-I system*. PEPMA report. Department of Computer Science, University of Bristol, 1991.

Vítor Santos Costa. *Compile-time analysis for the parallel execution of logic programs in Andorra-I*. Ph.D. thesis, Department of Computer Science, University of Bristol, 1993.

(The following remarks are based on their work.)

Disallowing early execution (sequencing)

- A goal is *sensitive* if is has side-effects or if its behaviour can be changed by early execution.

  atom(X) is sensitive; early execution could bind X where Prolog would not.

- Sequential conjunction is introduced to prevent early execution to the right of sensitive goals and ancestors of sensitive goals.

```
p(X) :- p1(X), atom(X) :: p2(X).

q(X) :- q1(X), p(X) :: q2(X).

...
```

- Abstract interpretation may establish that sufficient input is given for a goal to not be sensitive.

  If it is known that X will be bound by Prolog execution, atom(X) is not sensitive.

Establishing determinacy of goals

New goals are tested for determinacy.
If nondeterminate, they are suspended on suitable variables.

- Generalisation of indexing
- Decision trees / decision graphs
- More complex to deal with aliasing of input arguments
- Pruning – cut & commit

(See Santos Costa 1993.)

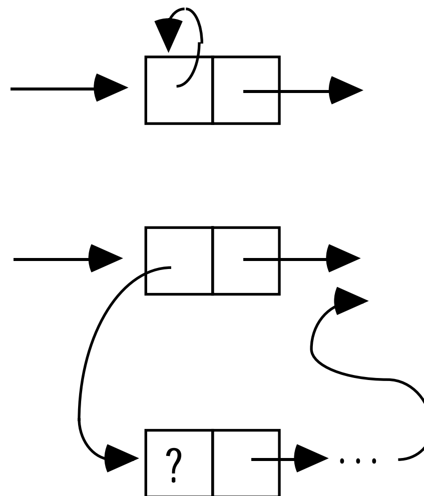
Maintaining order of goals

Unlike suspension of goals as by freeze in Prolog,
the order of suspended nondeterminate goals has to be maintained.

Simple solution: double-linked list

Better: semi double-linked list (Andorra-I)
(less locking & trailing)

Basic idea



(See Yang et al 1991.)

Sharing of goal structure

Upon creation of alternatives, different modifications to the list of goals are made in different branches.

Updates are trailed and backtracked.

Cells can be multiply assigned
(but only once per choice point)

(See Yang et al 1991.)

Beyond BAM for Prolog

IDIOM (Gupta, Costa, Yang, Hermenegildo)

- Integrates independent and-, dependent and-, and or-parallelism.
- Based on BAM and Gupta's Extended And-Or Tree Model.

Extended Andorra Model (EAM) (Warren)

- Allows nondeterminate and-parallel execution.
- Aims at extracting maximum parallelism as well as doing the best possible search.

(Not Yet Implemented)

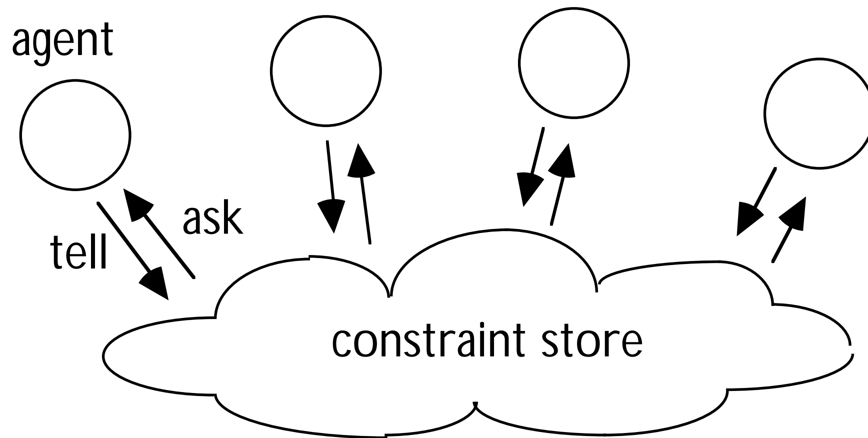
Conclusions

- BAM gives expressiveness and dependent and-parallelism.
- Prolog+BAM requires analysis: unpredictable for programming, useful for parallel execution.

CCP LANGUAGES

- Flat CCP
- Deep CCP
- Stability
- Higher-Order CCP

Flat CCP



Agents *tell* constraints to a shared constraint store and may *ask* if constraints are entailed by the store.

Michael J. Maher. Logic semantics for a class of committed-choice programs. In *Proceedings of 4th ICLP*, MIT Press, 1987.

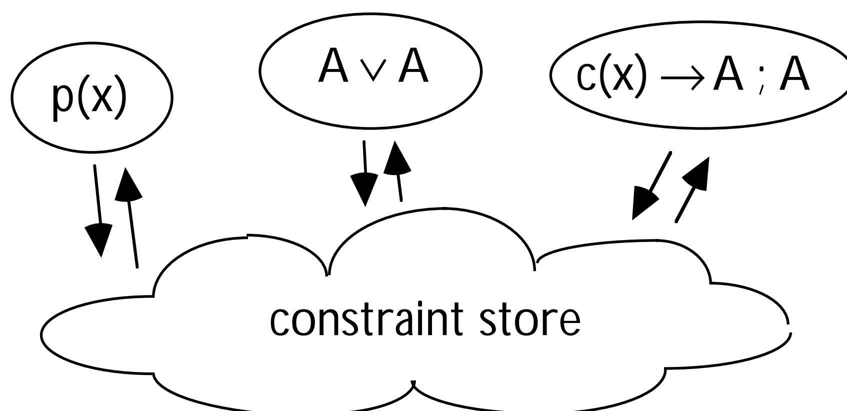
Vijay A. Saraswat. *Concurrent Constraint Programming Languages*. The MIT Press, 1993.

A flat language (cc flavour)

A	::=	c(x)	*constraint*
	|	p(x)	*call*
	|	A, A	*composition*
	|	x : A	*hiding*
	|	$A \vee A$	*disjunction*
	|	$c(x) \rightarrow A ; A$	*conditional*
D	::=	p(x) := A	*definition*

$x_1 : (A_1, c_1, (x_2 : A_2, c_2, A_3))$

$x_1, x_2 : \ c_1, c_2, \ A_1, A_2, A_3$



A flat language - mplistic rules

DETERMINATE

$p(x) \;\Rightarrow\; A[x/y]$ if $p(y) := A$

$\rightarrow A \,;\, B \;\Rightarrow\; A$

$\perp \rightarrow A \,;\, B \;\Rightarrow\; B$

$\perp, A \;\Rightarrow\; \perp$

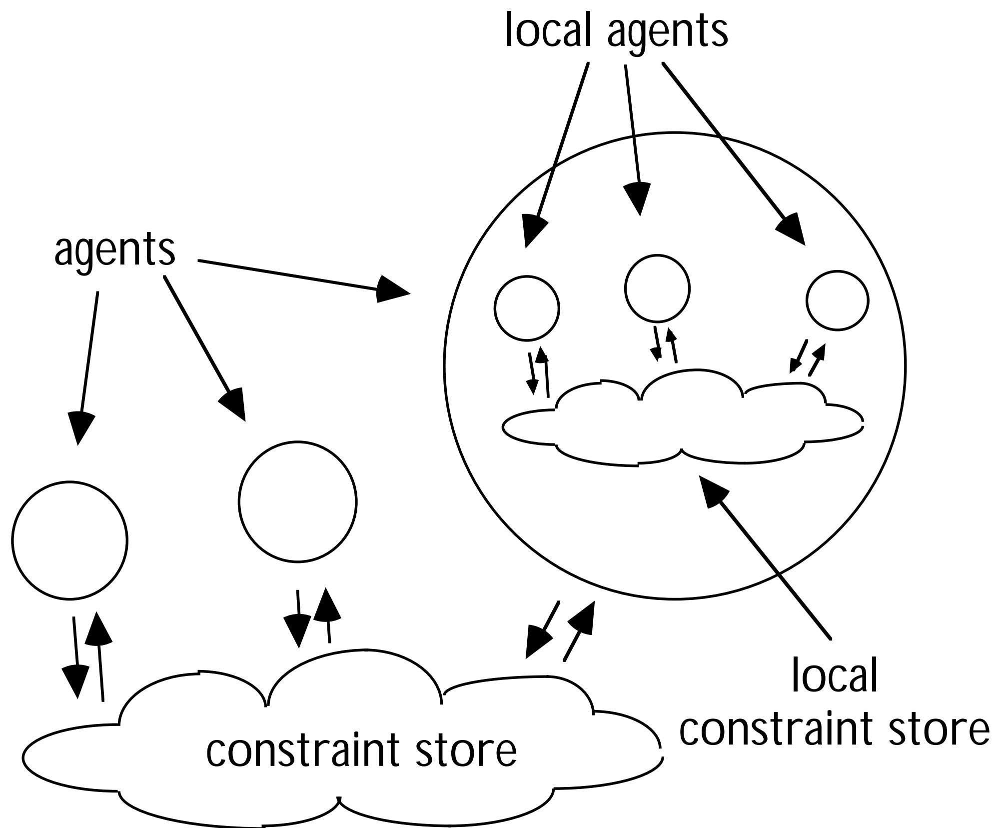
$(\perp, A) \vee B \;\Rightarrow\; B$

NONDETERMINATE

$(A \vee B), C \;\Rightarrow$

A, C and B, C

Deep CCP



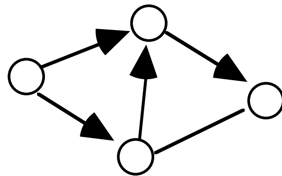
- Local agents see local and external constraints.
- Scope of nondeterminism is local.
- Enables encapsulated search with reactive top-level.
- Supports negation and general conditions.

Gert Smolka. *A calculus for higher-order concurrent constraint programming with deep guards*. RR-93-16. German Research Center for AI (DFKI).

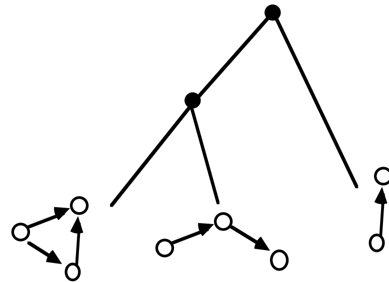
Sverker Janson. *AKL—A Multiparadigm Programming Language*. SICS Dissertation Series 14, Swedish Institute of Computer Science, 1994.

Flat vs deep

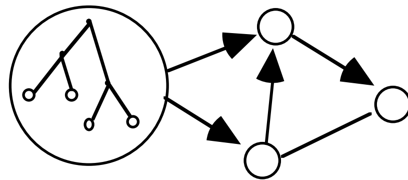
flat+det:



flat+nondet:



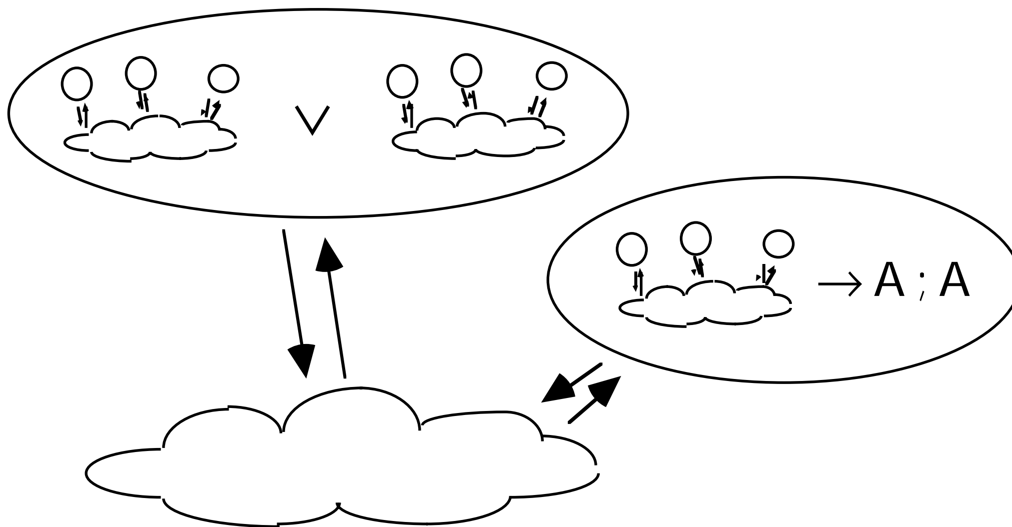
deep+nondet:



	"det"	nondet
flat	KL1, Janus, …	cc, Andorra-I, Pandora, …
deep	CP, GHC, Parlog, …	AKL, Oz

Deep CCP (simplistic)

A	::=	c(x)	*constraint*
	|	p(x)	*call*
	|	A, A	*composition*
	|	x : A	*hiding*
	|	$A \vee A$	*disjunction*
	|	$A \rightarrow A ; A$	*conditional*
D	::=	p(x) := A	*definition*



Deep CCP - simplistic rules

DETERMINATE

$p(x) \quad \Rightarrow \quad A[x/y] \qquad\qquad$ if $p(y) := A$

$(\quad \vee C) \rightarrow A \; ; B \quad \Rightarrow \quad A$

$\bot \rightarrow A \; ; B \quad \Rightarrow \quad B$

$\bot, A \quad \Rightarrow \quad \bot$

$\bot \vee B \quad \Rightarrow \quad B$

NONDETERMINATE

$(A \vee B), C \quad \Rightarrow$

$\qquad (A, C) \vee (B, C) \qquad$ if $((A \vee B), C)$ stable

Stability

Generalisation of the BAM. (Introduced with AKL.)

An expression A is *stable* if

> For all satisfiable constraints c,
> (c, A) is not reducible with determinate rules.

A nondeterminate step is allowed in a stable expression.

For example

$$x, y : \quad p \vee q, \quad x = a, \quad y = b \rightarrow r \; ; \; s$$

is stable

$$x : \quad p \vee q, \quad x = f(y)$$

is also stable

$$x : \quad p \vee q, \quad y = f(x)$$

is *not* stable.

AKL

A	::=	c(x)	*constraint*
	|	p(x)	*call*
	|	A, A	*composition*
	|	x : A	*hiding*
	|	$(C_\% ; \cdots ; C_\%)$	*choice*
	|	Σ(x, A, y)	*aggregation*
$C_\%$	::=	A % A | x : $C_\%$	*clause* (% = →, |, or ?)
D	::=	p(x) := A	*definition*

- The AGENTS implementation of SICS offers record constraints and FD constraints.
- Prolog-like builtins and programming environment.
- Speed at best 1/2 of emulated SICStus.
- Speed of FD part 1/2 of clp(FD).
- Release 1.0 in mid-fall.

Higher-Order CCP

A	::=	c(x)	*constraint*
	|	y(x) := A	*abstraction*
	|	y(x)	*application*
	|	A, A	*composition*
	|	x : A	*hiding*
	. . .		

- Subsumes λ-calculus.
- Elegant support for modules and objects.
- Enables exciting new view of search.

Oz

A deep-guard higher-order concurrent constraint programming language. (Smolka et al)

E	::=	**false** | **true** | $x = s$	*constraints*
	|	$x = l(y_1 \ldots y_n)$	*tuple construction*
	|	$x = l(l_1 : y_1 \ldots l_n : y_n)$	*record construction*
	|	**proc** $\{ x\ \overline{y} \}$ E **end**	*procedure definition*
	|	$\{ x\ \overline{y} \}$	*procedure application*
	|	$E_1\ E_2$	*concurrent composition*
	|	**local** $\overline{x}$ **in** E **end**	*variable declaration*
	|	**if** C_1 [] ... [] C_n **else** E **fi**	*conditional*
	|	**or** C_1 [] ... [] C_n **ro**	*disjunction*
	|	**process** E **end**	*process creation*
C	::=	$\overline{x}$ **in** E_1 **then** E_2	*clause*
x, y, z	::=	⟨variable⟩	
l	::=	x | ⟨atom⟩	
s	::=	l | ⟨number⟩	
$\overline{x}, \overline{y}$	::=	⟨possibly empty sequence of variable⟩	

Extension of Oz for search (simplified)

A ::= solve(x, A, y) *solver*

solve(x, $\bot$, y) $\Rightarrow$ y = fail

solve(x, ((A $\vee$ B), C), y) $\Rightarrow$

y = split(z_1, z_2),
(z_1(x) := A, C),
(z_2(x) := B, C) if ((A $\vee$ B), C) stable

solve(x, A, y) $\Rightarrow$

y = solved(z),
z(x) := A if A stable

C. Schulte, G. Smolka, and J. Würtz. Encapsulated Search and Constraint Programming in Oz. In *Principles and Practice of Constraint Programming 1994*.

IMPLEMENTATION OF DEEP CCP

- Basic issues
- Establishing stability
- Copying vs sharing
- Avoiding nondeterminism

[• Parallelism
(Moolenaar and Demoen;
Montelius and Ali)]

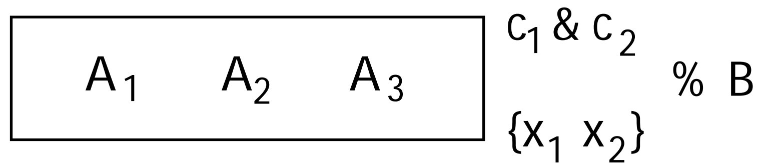
Basic issues

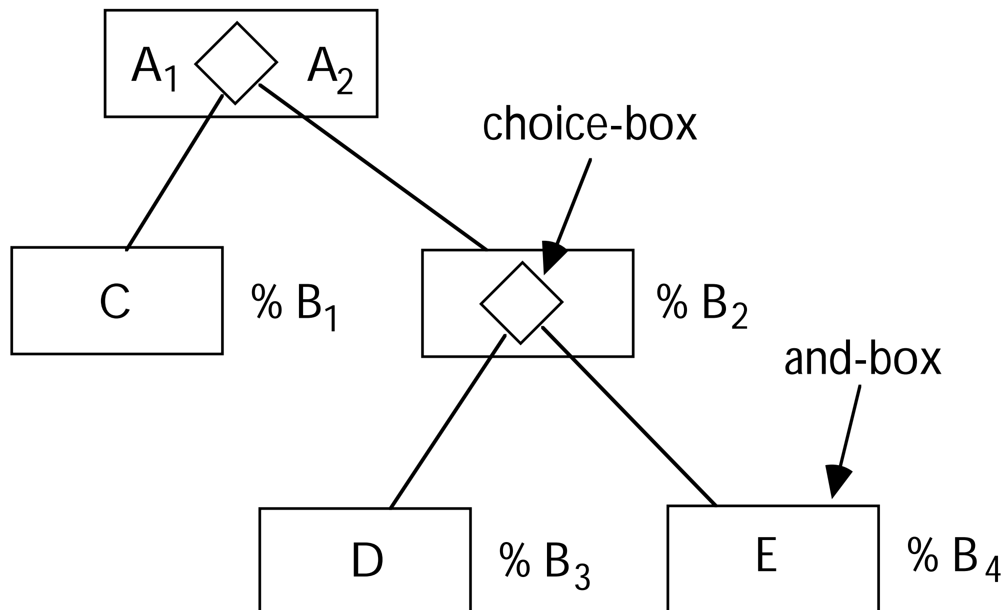
- Execution states
- Hierarchical constraint store
- Suspension and waking
- Execution strategy

Execution states

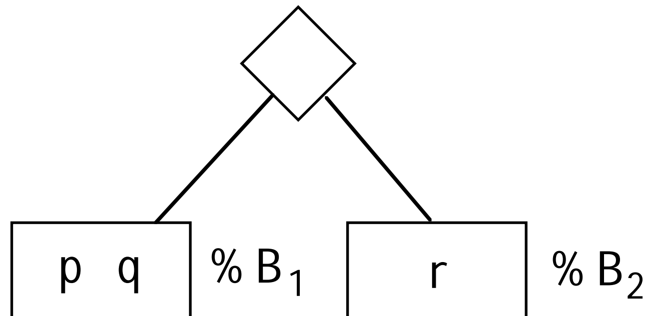
$x_1 : (A_1, c_1, (x_2 : A_2, c_2, A_3) \% B)$

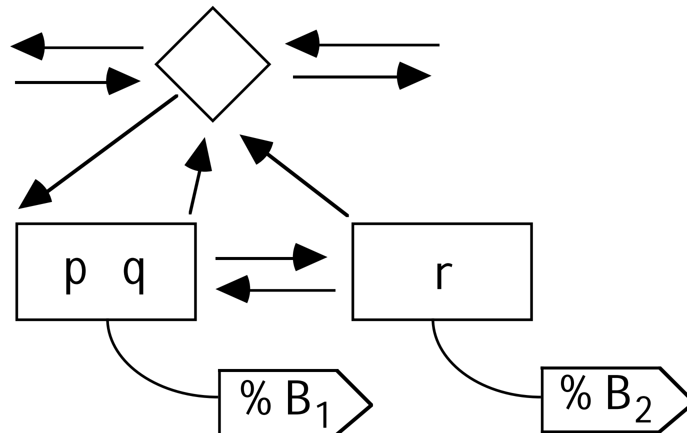
$x_1, x_2 : c_1, c_2, A_1, A_2, A_3 \% B$





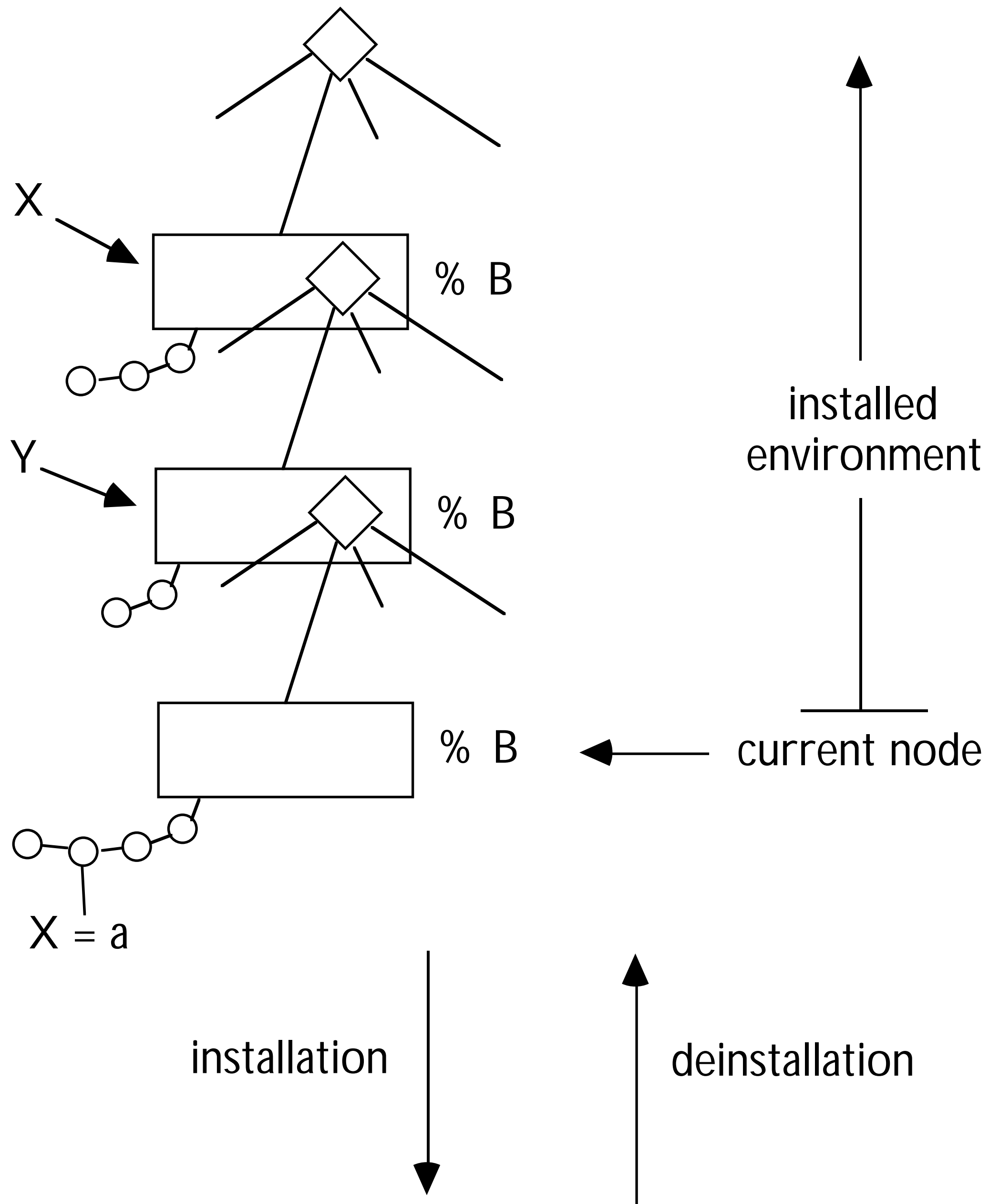
Representation



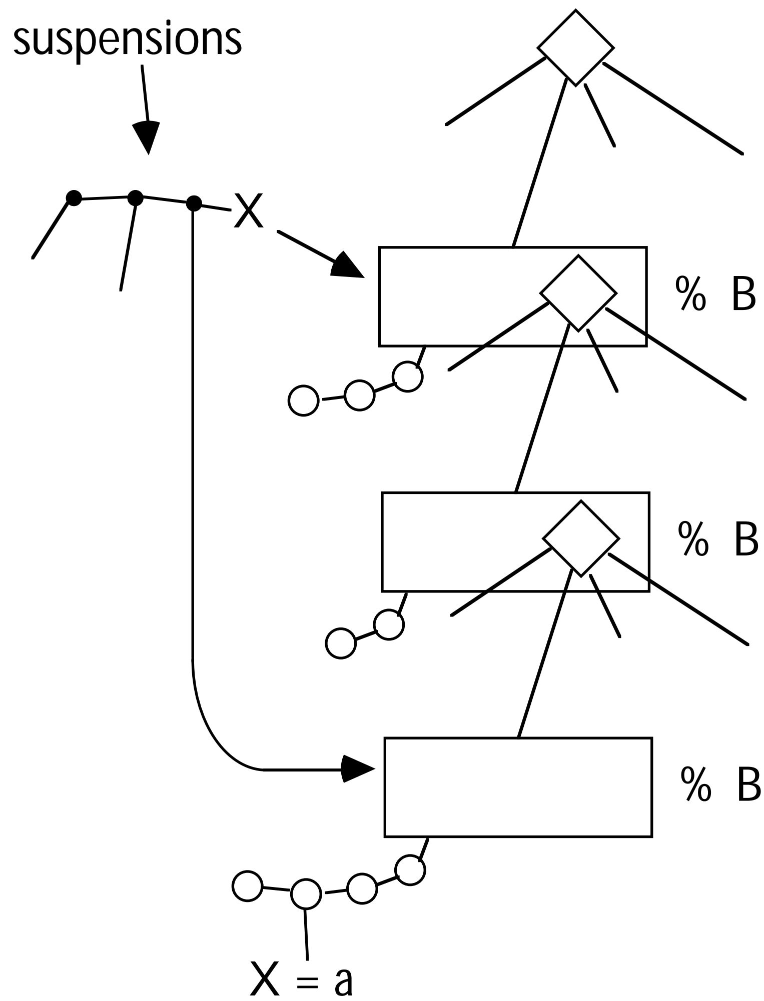


and-node	choice-node	continuation
parent alternatives constraints goals continuation forward state	parent goals alternatives	cont. ptr. y-registers ...

Hierarchical constraint store

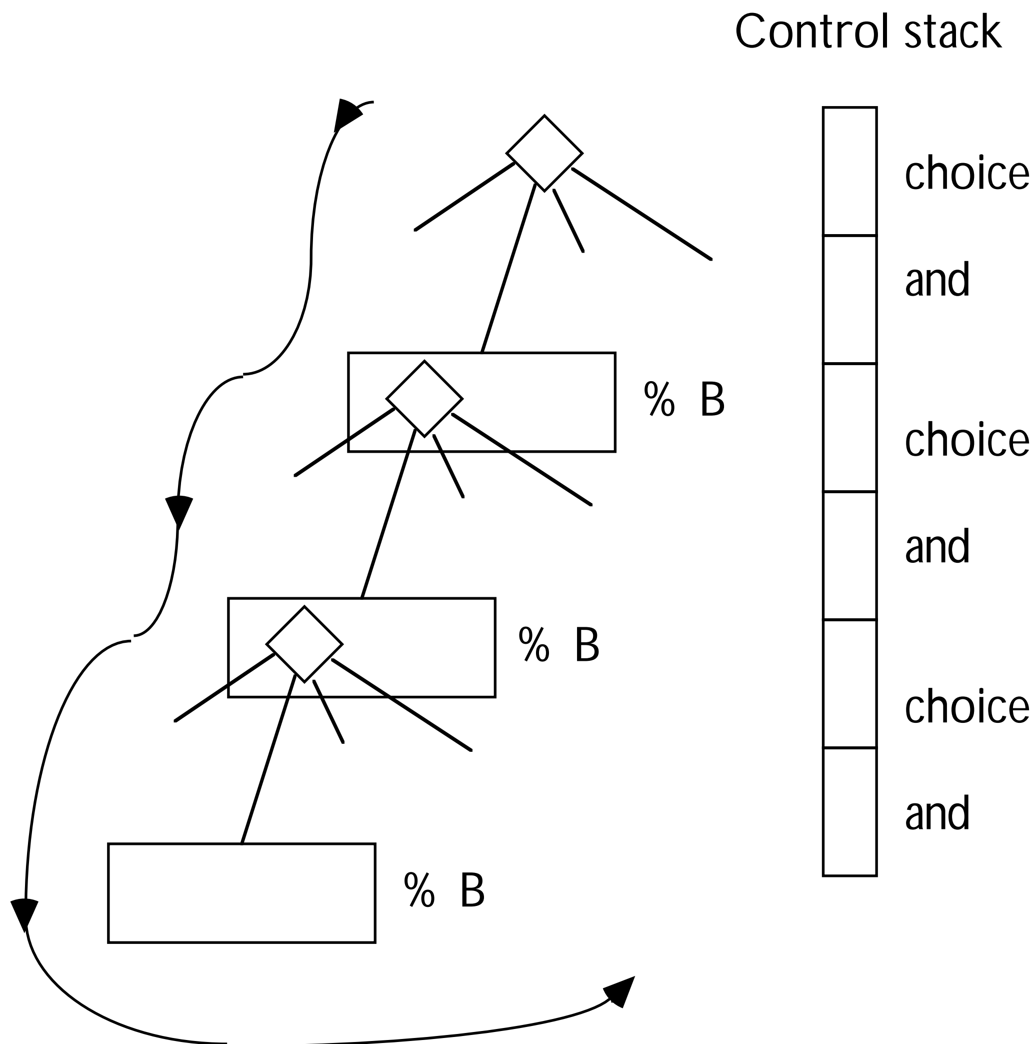


Suspension and waking



Only waking in the scope of a new binding!

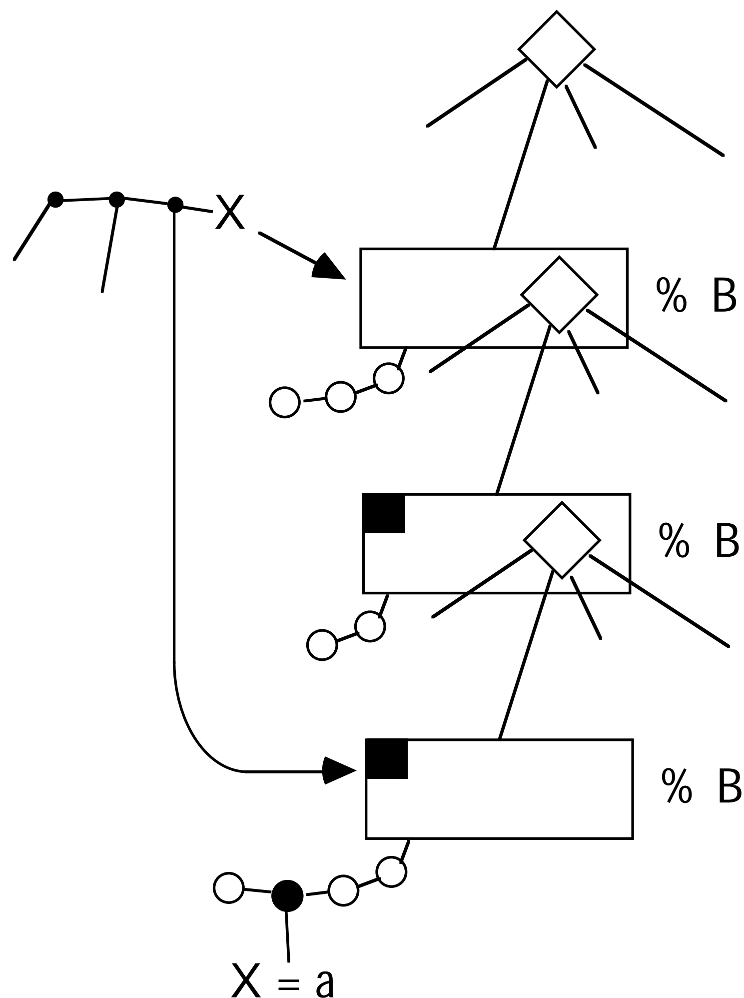
Execution strategy



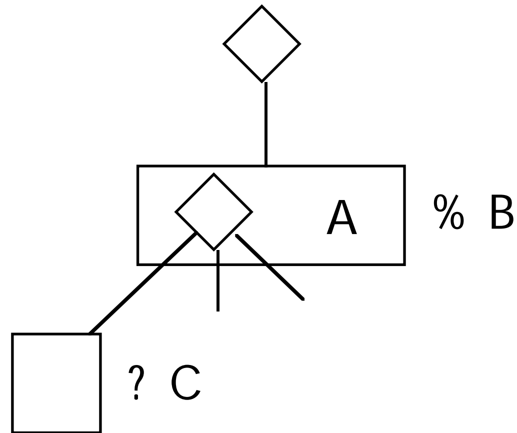
No need to share and-tasks the way environments are shared in Prolog! (Makes iteration a'la Meier simple.)

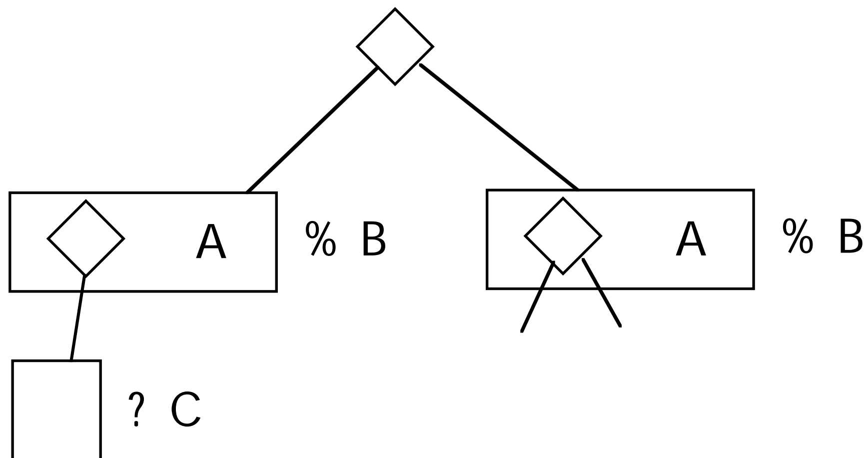
Stability

- Depth first execution enables simple detection of remaining determinate reductions.
- Independence of new constraints in environment = no suspensions on variables in environment.
- Information on suspensions is associated with ancestor and-boxes: flag, counter, or list of suspensions.



Copying vs sharing





- Cost of copying
- Simple sharing
- General sharing
- Labelling by backtracking

Cost of copying (a truly bad case)

bagof(X, tail(X, L), S)   (in AGENTS on DEC 5000/240)

Length	Total	% Copy	% GC	% Other
300	309	77	12	11.1
600	1098	76	18	5.8
900	2518	75	22	2.9
1200	4455	77	20	3.1
1500	7268	76	21	2.4
1800	12293	69	29	2.2

tail(X, L), fail

findall(X, tail(X, L), S)   (in SICStus on DEC 5000/240)

Length	Failure	Findall
300	3.3	334
600	6.0	1307
900	9.6	2877
1200	12.9	5048
1500	15.6	7927
1800	18.6	11475

Cost of copying (better cases)

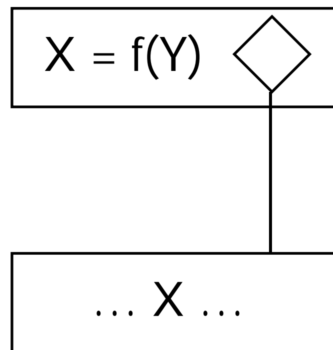
Copying performs better for constraint solving:

Example	Total	%Cp	%GC	%Oth	Cp:s
queens(4)	16	25	0	75	1
queens(8)	89	35	8	57	4
queens(16)	608	47	20	33	8
money(dif)	260	66	3	31	152
zebra(cir)	88	62	6	32	29
substitution	1758	67	15	18	182
frequency	229	55	21	24	45
knights(5)	131	29	5	66	24
knights(6)	973	53	8	39	201
allknights(5)	15870	25	6	69	3213

Why copying?

- Simpler!
- No overhead when not used!
- Simplifies memory management!

Simple sharing

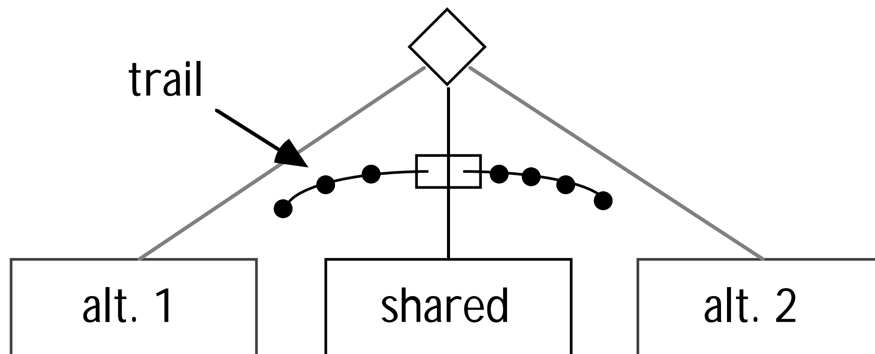


- Add home pointer to “structs”.
- Do not copy external “structs”.

E.g., in bagof(X, tail(X, L), S), L is external to bagof:

Length	Total	% Copy	% GC	% Other
300	59	57	0.0	43
600	117	55	2.7	48
900	166	54	3.7	42
1200	228	54	3.3	43
1500	290	51	5.6	43
1800	366	52	9.8	39

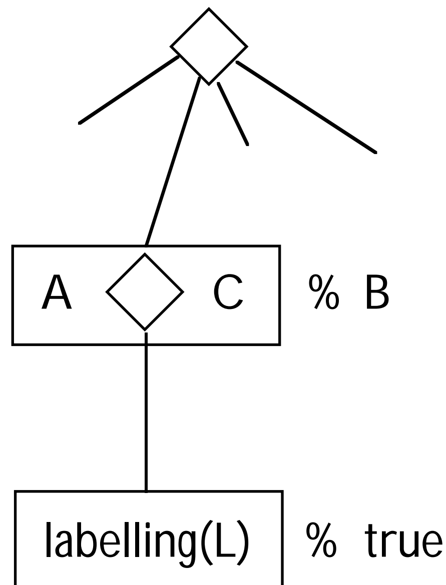
General sharing



- Associate time stamps with updatable objects.
- Introduce sharing nodes that branch “and-boxes”, and contain value trails for the two branches.
- Value trails are installed and deinstalled like constraints.
- Sharing nodes have time stamps higher than all previous objects.
- New objects are given time stamps higher than any ancestor sharing nodes.
- Changes to objects older than some ancestor sharing node are trailed in any younger ancestor sharing node.

Non-suspending execution without trailing!

Labelling by backtracking



For FD, the deep labelling idiom can be optimised, using backtracking as in "plain" CLP.

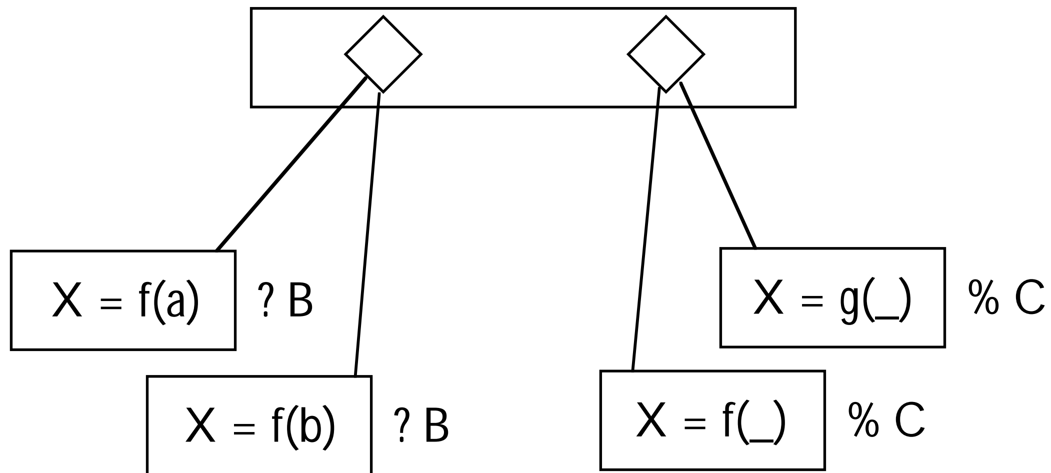
Program	AKL/Cp	AKL/Bt	CLP(FD)	ECLiPSe
16-queens	8210	2320	1140	9720
all 8-queens	1060	290	190	1320
alpha ff	810	240	120	2300
eq20	1200	500	220	750
cars	260	90	50	130

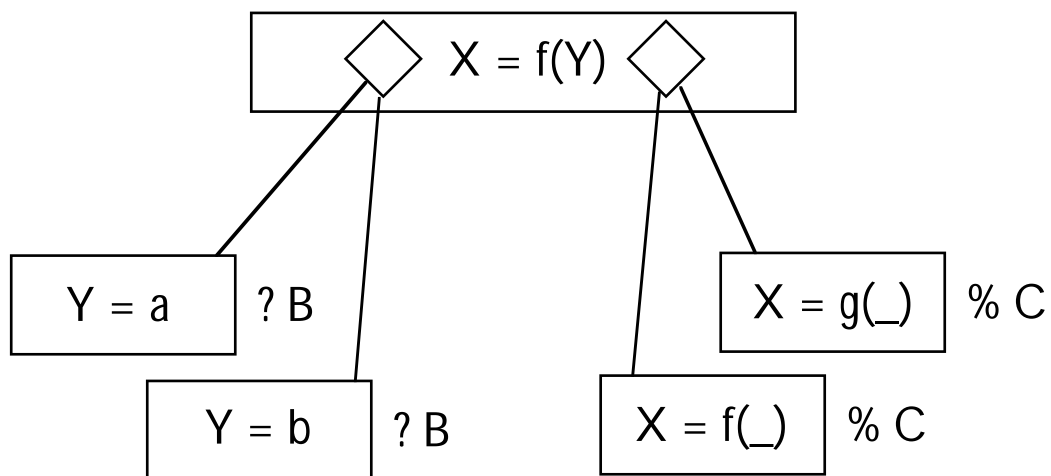
(Timings on the same SPARC-2 system.)

Avoiding nondeterminism

- Constraint lifting / constructive disjunction
- Consistency techniques
- Better choice of candidates

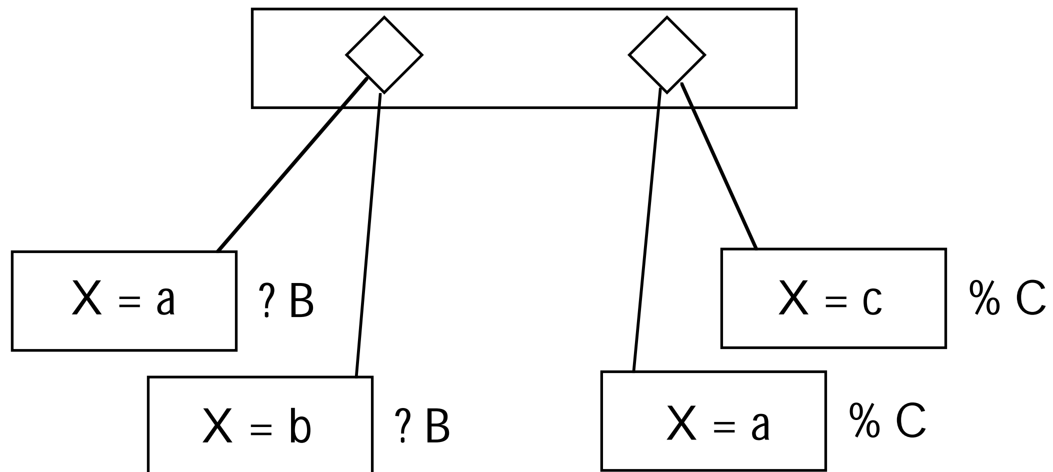
Constraint lifting / constructive disjunction





(For FD: Van Hentenryck, Saraswat, Deville 92)

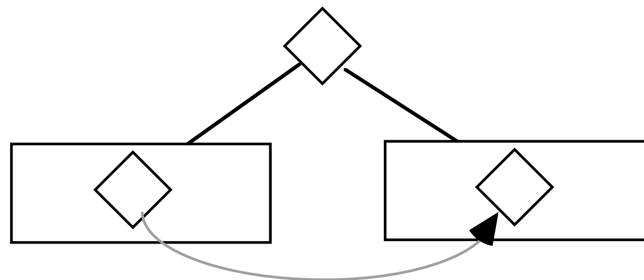
Consistency techniques



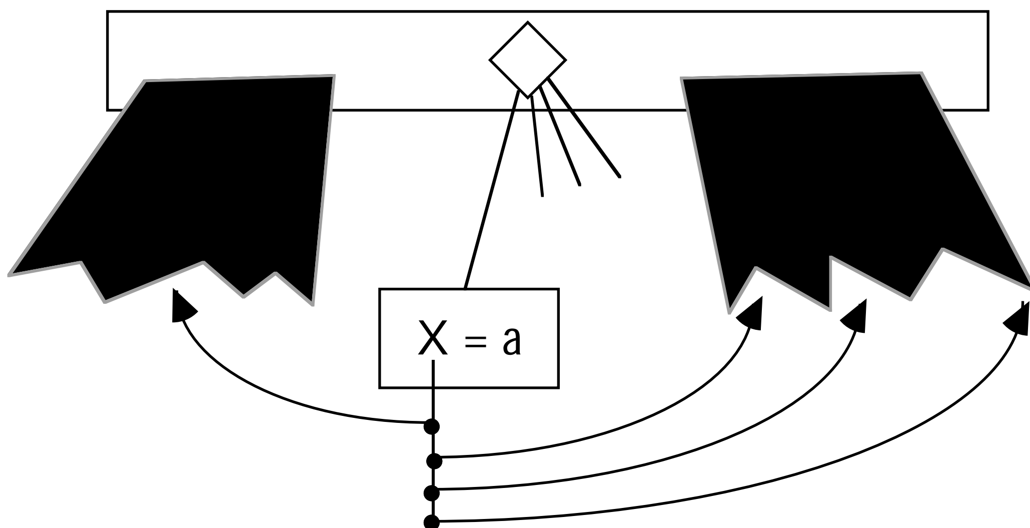
(Moolenaar and Demoen, this conference)

Better choice of candidates

- When a branch fails, it might be advantageous to try the failing goal earlier subsequent branches. (Abreu, Pereira, Codognet JICSLP92)



- If a candidate binds more variables, or there are more suspensions on the variables bound, it is likelier to lead to failure. (Moolenaar and Demoen, this conference)



Conclusions

- Deep CCP is a rich setting for clever implementation techniques.
- Efficiency need not be sacrificed.
- Much is unexplored…

  (E.g.
  - general sharing
  - avoiding/minimising copying
  - relation to H-O solve combinator)